Chapter 1
Object Oriented Programming Approach

The term Object Oriented programming is frequently heard in the programming arena. Object oriented approach was started to overcome the limitations of the earlier programming approaches. It is popularly known by its acronym OOP. It is used to develop reliable and reusable software.

The programming technology is continuously developing since the start of the computer and related technologies. New tools and techniques are included in programming in each phase of their development. Such enhancements increased complexity in programming and design of large software. Similarly, the users' requirements change and increase after the software is brought into operation. The software may need maintenance and enhancements after the regular feedback from the users. There could be problems to represent real life entities of problem while analyzing and designing system. While improving the software work may need to begin from the scratch that may increase software cost too. To incorporate users' demands and enhancements in software with such complex systems was difficult. To overcome such problems software developers were forced to develop new programming method. OOP was introduced to solve such programming problems.

1.1 Software Evolution

The software evolution occurred in several phases. Since the beginning of the first computer, programming for the computer started to develop software. The earlier electronic computer ENIAC was programmed in machine language by using switches to enter 1 and 0.

1.2 Basic of Object Oriented Programming

Objects are the entities that can be uniquely identified from others. They have their unique identity and found everywhere. In real world system everything exists in the form of objects. For example desk, bench, blackboard, student, teacher, car, tree are objects. Every object has two things, firstly its properties we call attributes and second its behavior we call function. For example a car is an object. It has attributes like color, number of seats, chassis number, engine number etc and behavior like move, stop, accelerate, turn etc. The Object Oriented Programming is developed to model such real world system. Its sole objective is to overcome the limitation of Procedure Oriented approach. Before discussing various features of Object Oriented Programming, it is wise to discuss characteristics of Procedure Programming and its limitations.

1.2.1 Procedure Oriented Programming

In procedure oriented programming a large program is broken down into smaller manageable parts called procedures or functions. In procedure oriented programming priority is given on function rather than data.
In procedure oriented programming language, a program basically consists of sequence of instructions each of which tells the computer to do something such as reading inputs from the user, doing necessary calculation, displaying output. When a program becomes

larger, it is then broken into smaller units called procedure or functions. A number of functions are supposed to be written to accomplish such tasks. The primary focus of procedural oriented programming is on functions rather than data. These functions do not let code duplication. This technique is only suitable for medium sized software applications.
The procedure oriented programming can be diagrammatically represented as follows:

In procedure oriented programming two types of data local and global are used. Data within the function are called local data and the data which are not within any function are called global data. Global data are accessible to the only function where it is declared. So each function may access its local data as well as global data. The local data of one function is not accessible to other functions. If any data is to be accessed by two or more functions it should be made global. However, global data are vulnerable to another programmer to be changed unknowingly. Functions are action oriented and do not correspond to the element of the problem. The separate arrangement of data and functions does a poor job of modeling things in the real world. That's why procedure oriented programming approach does not model real world system perfectly.

High Level Programming Languages like COBOL, FORTRAN, Pascal, C are common procedure oriented programming languages.

Characteristics of POP

The characteristics of procedure oriented programming are listed as follows:

- A large program is broken down into small manageable procedures or functions.
- Procedure oriented programming focuses on procedure or function rather than data.
- For sharing a common data among different functions the data is made global.
- Since global data are transferred from function to function; during the course of transformation the global data may be altered by the function.
- The program design of procedure oriented programming follows top down methodology.

Limitation of POP

Even though procedure oriented programming approach is still used in software industry it has following limitations.

- Focus on functions rather than data.
- In large program, it is difficult to identify belonging of global data.
- The use of global data is error prone and it could be an obstacle in code maintenance and enhancements.
- The modification of global data requires the modification of those functions using it.
- Maintaining and enhancing program code is still difficult because of global data.
- It does not model real world problem very well. Since functions are action oriented and do not really correspond to the elements of problem.

Object oriented programming (OOPS)

The errors faced in the procedure oriented programming approach are the motivating factor in the invention of objected oriented approach. In OOP, data are treated as a critical element in the program and restricts freely transformation of data around the system. Instead, data are associated with functions that operate on it and protect it from accidental modification outside functions. OOP approach permits decomposition of a problem into entities called objects and then build data and function around them. Data of an object are accessible only by the function belonging with the object. But function of one object may access the function of another object.

Object-Oriented programming is a programming methodology that associates data structures with a set of operators which act upon it. In OOP, an instance of such an entity is known as object. In other words, OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.

Characteristics of OOPs

OOP is most sophisticated programming methodology among other methodologies by far. Some noticeable characteristics of OOP are as follows:

- Emphasis is on data rather than procedures.
- Programs are divided into objects.
- Function and data are tied together in a single unit.
- Data can be hidden to prevent from accidental alteration from other function or objects.
- Data access is done through the visible functions so that communication between objects is possible.
- Data structures are modeled as objects.
- Follows Bottom up approach of program design methodology.
-

Procedure oriented versus Object oriented programming

The differences between procedural and Object oriented programming are tabulated below:

Procedure Oriented Programming	**Object Oriented Programming**
Emphasis is given on procedures.	Emphasis is given on data.
Programs are divided into functions.	Programs are divided into objects.
Follow top-down approach of program design.	Follow bottom-up approach of program design.
Generally data cannot be hidden.	Data can be hidden, so that non-member function cannot access them.
It does not model the real world	It models the real world problem

problem perfectly.	very well.
Data move from function to function.	Data and function are tied together. Only related function can access them
Maintaining and enhancing code is still difficult.	Maintaining and enhancing code is easy.
Code reusability is still difficult.	Code reusability is easy in compare to procedure oriented approach.
Examples: FORTRAN,COBOL, Pascal, C	Example: C++, JAVA, Smalltalk

Features of Object Oriented programming

Different features of object oriented programming are explained here.

1. Object

Objects are the entities in an object oriented system through which we perceive the world around us. We naturally see our environment as being composed of things which have recognizable identities & behavior. The entities are then represented as objects in the program. They may represent a person, a place, a bank account, or any item that the program must handle. For example Automobiles are objects as they have size, weight, color etc as attributes (that is data) and starting, pressing the brake, turning the wheel, pressing accelerator pedal etc as operation (that is functions).

.

Following are some of the examples of objects in different scenario.

a) Physical Objects

- Bus in Traffic System
- Atom in chemical composition
- Diode in electronic system
- Humidity in metrological system
- Leader in political syste

b) Graphical User Interface

- Menu
- Button
- Toolbar
- Combo box
- Text box
- Windows

e) Geometrical Shapes

- Point
- Line
- Triangle
- Circle
- Ellipse

f) User defined data

- Distance
- Currency
- Time
- Date
- Complex Number

In computer programming, all these objects of the real world can be modeled by combining data and function together to make object of the program which is not possible in procedure oriented programming.

2. Class

Object consists of data and function tied together in a single unit. Functions are used to manipulate on the data. The entire construct of objects can be represented by a user defined data type in programming. The class is the user defined data type used to declare the objects. Actually objects are the variable of the user defined data type implemented as class. Once a class is defined, we can create any number of objects of its type. Each object that is created from the user defined type implemented as class is associated with the data type of that class. For example, manager, peon, secretary clerk are the objects of the class employee. Similarly, car, bus, jeep, truck are the objects of the class vehicle. Classes are user defined data type (like a struct in C programming language) and behave much like built in data type (like int, char, float) of programming language. It specifies what data and functions will be included in objects of that class. Defining class doesn't create an object; however defining process specifies the data and function to be in the objects of its type.

One of the objects of student can have following values

Name = "Bishal"
Registration_number = 200876255
Marks = {66, 77, 51, 48, 82}

The function Sort_name () will sort and display list of students on the basis of name in alphabetical order. Similarly, function Tot_marks () will sum the marks obtained by the student. The function Percentage_marks() will calculate the percentage and the Decide_division () function will decide division based on percentage obtained by the student.

Each class describes a possibly infinite set of individual objects, each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class. The following points give the idea of class:
A class is a template that specifies data and their operations.
A class is an abstraction of the real world entities with similar properties.
Ideally, the class is an implementation of abstract data type.

3. Abstraction

Abstraction is representing essential features of an object without including the background details or explanation. It focuses the outside view of an object, separating its essential behavior from its implementation.

We can manage complexity through abstraction. Let's take an example of vehicle. It is constructed from thousands of parts. The abstraction allows the driver of the vehicle to drive without having detail knowledge of the complexity of the parts. The driver can drive the whole vehicle treating like a single object.

Similarly Operating System like Windows, UNIX provides abstraction to the user. The user can view his files and folders without knowing internal detail of Hard disk like the sector number, track number, cylinder number or head number. Operating System hides the truth about the disk hardware and presents a simple file-oriented interface.

The class is a construct in object oriented programming for creating user-defined data for abstraction. When data and it operation are presented together, the construct is call ADT (Abstract Data Type). In OOP classes are used in creating ADT. For example, a student class can be made and can be available to be used in programs. The programmer can implement the class in creating objects and its manipulation without knowing its implementation. The program can use the function Sort_name() to sort the names in alphabetical order without knowing whether the implementation uses bubble sort, merge sort, quick sort algorithms.

4. Encapsulation

The mechanism of wrapping up of data and function into a single unit is called encapsulation. Because of encapsulation data and its manipulating function can be kept together. We can assume encapsulation as a protective wrapper that prevents the data being accessed by other code defined outside the wrapper. By making use of encapsulation we can easily achieve abstraction.

The purpose of a class is to encapsulate complexity. Each data or function in a class can be marked as private or public. The public interface of a class represents everything that external users of the class may know about the data and function. The private function and data can only be accessed by code that is a member of a class. The code other than member of a class cannot access a private function or data. This insulation of data from direct access by the program is called data hiding. After hiding data by making them private, it then safe from accidental alteration.

The public interface should be carefully designed no to expose too much of the inner working of a class.

5. Inheritance

Inheritance is the process by which objects of one class acquire the characteristics of object of another class. 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 provides the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the existing code. We naturally view the

whole world is made up of objects. Many objects are related to each other in a hierarchical way, such as vehicle, four wheeler, and car. If we describe vehicle in an abstract way, the attributes may be such as color, number of seats etc. All vehicles have common behavioral aspect like; they move, accelerate, turn and stop. The more specific class of vehicle is four wheeler that acquires all features of class vehicle and has more specific attributes like engine number, chases number etc. The class vehicle is called base class (or super class) and class four wheeler is called derives class (or subclass).

6. Reusability

Like library functions in procedural programming a class in Object Oriented Programming can be distributed for further use. In OOP, the concept of inheritance provides the idea of reusability. Once a class is completed and tested, it can be distributed for the development other programs too. The programmer can add new features or make some changes or can derive new classes from the existing class. This idea saves time and effort of a programmer. The testing of software will become easier as the already tested class should not be tested again. Suppose we have got a tested class Employee and we have to design a new class for Manager. The class Manager has all common features to class Employee. We can add some more features to class Manager using all features of class Employee.

If a software company creates generic classes for one project then the company can use the same class and its extensions in the new project with less time, effort and investment.

7. Polymorphism

Polymorphism means 'having many forms'. The polymorphism allows different objects to respond to the same operation in different ways, the response being specific to the type of object. The different ways of using same function or operator depending on what they are operating on is called polymorphism.

Example of polymorphism in OOP is operator overloading, function overloading. Still another type of polymorphism exist which is achieved at run time also called dynamic binding.

For example operator symbol '+' is used for arithmetic operation between two numbers, however by overloading (means given additional job) it can be used over Complex Object like currency that has Rs and Paisa as its attributes, complex number that has real part and imaginary part as attributes. By overloading same operator '+' can be used for different purpose like concatenation of strings.

When same function name is used in defining different function to operate on different data (type or number of data) then this feature of polymorphism is function overloading.

8. Dynamic binding

The linking of a function call to the code to be executed in response to the call is called binding. There are two types of binding one is static binding(also called early binding) and another is dynamic binding(also called late binding). Function overloading and operator overloading construct in OOP are the examples of early binding. The early

binding occurs at the compile time. This type of polymorphism occurring at compile time is called compile time polymorphism.

Dynamic binding means that the code associated with a given function call is not known until the time of the call at run time. It is achieved at run time so called as run time polymorphism. Dynamic binding is possible only when we use inheritance and access the objects through pointers. If classes Circle, Box, and Triangle are derived from same function draw(). During the function call draw() through the pointer variable an appropriate function belonging to that class is involved.

9. Message passing

Procedural programming languages have function driven communication. That is a function is invoked for a piece of data. Object oriented language have message driven communication. A message is sent to an object. Communications among the objects are analogous to exchanging messages among people.

An Object-Oriented program consists of set of objects that communicate with each other. Object communicates with each other by sending and receiving message (information). A message for an object is a request for execution of a procedure and therefore will invoke a function or procedure in receiving object that generate the desired result. Message passing involves specifying the name of the object name of the function (message) and the information (arguments to function) to be sent. In word, the message for an object is a request for the execution of a function belonging to an object which generates the desired result for the given argument.

```
student.fee  (name) ;
object   message    information
```

Communication between the objects takes place as long as their existence. Objects are created and destroyed automatically whenever needed. In above example student is regarded as an object sending the message fee to find the fee to be paid by the student with the given name.

Popular Object oriented languages

An object-oriented programming language is one that follows object-oriented programming techniques such as encapsulation, inheritance, polymorphism. Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs, in which objects were the most important information to be represented. In around 1972 to 1980, a pure object-oriented programming language Smalltalk was developed. It was called pure object oriented language because everything in them was treated as objects.

It was designed specifically to facilitate and enforce Object Oriented methods.

Some languages like Java, Python were designed mainly for Object Oriented programming along with some procedural elements.

Apart from this, some languages like C++, Perl are historically procedural languages, but have been extended with some Object Oriented features.

Besides these, there are some languages that support abstract data type but not all features of object-oriented programming. They are called object-based languages. Examples of such language are Modula-2, Pliant, Ada.

1. Smalltalk

Smalltalk is an object-oriented, dynamically typed, reflective programming language The development of Smalltalk language started in 1969 and it was publicly available in 1980. This language was developed by Alen Kay, Dan Ingalls, Adele, Goldberg at Xerox Palo Alto Research center (PARC). This language is 100% Object Oriented. The development of this language is influenced by language like Lisp, Simula, Logo, and Sketchpad. ANSI Smalltalk was ratified in 1998 and represents the standard version of Smalltalk.

In smalltalk, objects are called instance variables. All objects are dynamic. It offers fully automatic garbage collection and deallocation is performed by a built in garbage collector. All variables are untyped and can hold objects of any class. New objects are created using the same message passing mechanism used for operations on objects. All attributes are private to the class where as all operations are public. The syntax is very unusual and this leads to learning difficulties for programmers who are used to conventional language syntax.

Inheritance can be achieved by supplying the name of the super class. All attributes of super class are available to all its descendants. All methods can be overridden. Also multiple inheritance is not supported by standard implementation of Smalltalk. Rapid development of program is possible under its highly interactive environment.

Example Program of Smalltalk:

```
Transcript show: 'Hello, world!'
```

In the above code, the message 'show:' is sent to the object 'Transcript' with the String literal 'Hello, world!' as its argument. Invocation of the 'show:' method causes the characters of its argument (the String literal 'Hello, world!') to be displayed in the transcript ('terminal') window.

2. Java

Java was designed by SUN (Stanford University Net) Microsystems, released in 1996 and is a pure object oriented language. The SUN says "Java is a new, simple, object oriented, distributed, portable, architecture natural, robust, secure, multi-threaded, interpreted, and high performance programming language".

It took 18 months to develop the first working version. Java was initially called "Oak". It was renamed Java in 1995. The objective of Java was "Write Once, Run Anywhere" (WORA). It was fairly secure and its security was configurable, allowing network and file access to be restricted. Major web browsers soon incorporated the ability to run secure Java applets within web pages. Java became popular quickly. With the advent of Java 2, new versions had multiple configurations built for different types of platforms. For example, J2EE was for enterprise applications and the greatly stripped down version J2ME was for mobile applications. J2SE was the designation for the Standard Edition. In

2006, for marketing purposes, new J2 versions were renamed Java EE, Java ME, and Java SE, respectively.
In 1997, Sun Microsystems approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a de facto standard that is controlled through the Java Community Process. At one time, Sun made most of its Java implementations available without charge although they were proprietary software. Sun's revenue from Java was generated by the selling of licenses for specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) which is a subset of the SDK, the primary distinction being that in the JRE, the compiler, utility programs, and many necessary header files are not present.
On 13 November 2006, Sun released much of Java as free software under the terms of the GNU General Public License (GPL). On 8 May 2007 Sun finished the process, making all of Java core code open source, aside from a small portion of code to which Sun did not hold the copyright.
The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode which can run on any Java virtual machine (JVM) regardless of computer architecture.
One characteristic, platform independence, means that programs written in the Java language must run similarly on any supported hardware/operating-system platform. One should be able to write a program once, compile it once, and run it anywhere.This is achieved by most Java compilers by compiling the Java language code halfway (to Java bytecode) which means simplified machine instructions specific to the Java platform. The code is then run on a virtual machine (VM), a program written in native code on the host hardware that interprets and executes generic Java bytecode. Further, standardized libraries are provided to allow access to features of the host machines (such as graphics, threading and networking) in unified ways. Note that, although there is an explicit compiling stage, at some point, the Java bytecode is interpreted or converted to native machine code by the JIT (Just In Time) compiler.

Example Program of Java:

```
// Hello.java
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

3. C# (C sharp)

C# is an object-oriented programming language developed by Microsoft as part of the .NET fromework and later approved as a standard by ECMA and ISO. Anders Hejlsberg leads development of the C# language, which has a procedural, object-oriented syntax based on C++ and includes aspects of several other programming languages most notably Delphi and Java with an emphasis on simplification. It was develop by Microsoft in 2000

and was standarderised by ECMA (European Computer Manufactor Association) in 2003.

Example Program of C#:

```
class ExampleClass
{
   static void Main()
   {
      System.Console.WriteLine("Hello, world!");
   }
}
```

4. C++

The programming language C++ was developed by Bjarne Stroustrup at Bell Lab in New Jersey in early 1980 s as extension of C. He named 'C with Classes'. In 1983 it was renamed to C++. The operator ++ meaning that increment in C. ++ is increment operator in C/C++.Enhancements started with the addition of classes, followed by, among other features, virtual functions, operator overloading, multiple inheritance, templates, and exception handling. The C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998, the current version of which is the 2003 version, ISO/IEC 14882:2003. A new version of the standard known informally as C++0x is being developed

C++ is a general-purpose programming language. C++ is regarded as a mid-level language, as it comprises a combination of both high-level and low-level language features. It is a statically typed, free-form, multi-paradigm, usually compiled language supporting procedural programming, data abstraction, object-oriented programming, and generic programming. The C++ language corrects most of the deficiencies of C by offering improved compile time type checking and support for modular and object oriented programming. C++ supports multiple inheritance and does not have garbage collector dynamically created object must be destroyed explicitly.

Example Program of C++:

```
#include <iostream>
using namespace std;
int main()
{
  cout<<'Hello, World!');
  return 0;
}
```

Advantages of OOP

Object oriented programming contributes greater programmer productivity, better quality of software and lesser maintenance cost. The main advantages are:

- Redundant code is eliminated by various techniques like inheritance templates.

- Through data hiding, programmer can build secure programs that cannot be invaded by code in other pats of the program.
- Existing classes can serve as library class for further enhancements. Classes are also available as library class in the standard library of the language.
- Because of division of program into objects makes software development easy.
- Software complexity is less severe than conventional programming techniques.
- Because of dynamic binding, addition of new classes of objects at run time is possible without modifying the existing code.
- The limitation realized in base class can be fulfilled in derived class without writing even a single piece of code in the base class.
- Upgrading and maintenance of software is easily manageable.
- System can be easily upgraded from small to large systems.
- Message passing technique makes the interface simpler with external systems.
- Models real world system perfectly.
- Code reusability is much easier than conventional programming languages.

Disadvantages of OOP

- Compiler and runtime overhead. Object oriented program required greater processing overhead demands more resources.
- An object's natural environment is in RAM as a dynamic entity but traditional data storage in files or databases
- Re-orientation of software developer to object-oriented thinking.
- 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.

Basic program constructs in C++

Introduction to C++

C++ is an object oriented programming language. It was developed by Bjarne Stroustrup at AT and T Bell Lab USA. It was developed on the base of C programming language and the first added part to C is concept of ***Class***. Hence originally this language was named "C with class". Later name was changed to C++ on the basis of idea of increment operator available in C. since it is the augmented or incremented version of C, it is named C++.

The first C++ program

Following is a sample program in C++ that prints a string on the screen(monitor).

```
#include<iostream.h>
int main()
{
        cout<<"Hi Everybody";
        return 0;
```

}

Like in C, functions are the basic building block in C++. The above example consists of a single function called main ().

When a C++ program executes, the first statement that executes will be the first statement in main () function. The main function calls member functions of various classes (using objects) to carry out the real work. It may also call other stand-alone functions.

In C++, the return type of the main function is 'int'. So, it returns one integer value to the operating system. Since the return type int is default, the keyword 'int' in main () is optional. So,

```
main ()
{
   ……..// also valid
}
```

Compilers generate error or warning if no value is returned. Many operating systems test the return values. If the exit value is zero (0), the operating system will understand that the program ran successfully. If the returned value (exit value) is non zero, it would mean that there was problem.

Comment syntax

In C++, comments start with a double slash (//) symbol and end at the end of the line. A comment may start at the beginning of a line or anywhere in the line and whatever follows till the end of that line is ignored. There is no closing symbol. If we need to comment multiple lines, we can write as

```
// this is an
// example
// of multi line comments
```

The C comment style /*…….*/ may also be used for multi line comments.

The output operator (Output using "*cout*")

The statement: *cout<<"Hi Everybody";* in the above example prints the phrase in quotation marks on the screen.

Here, the identifier 'cout', pronounced as "see out", is a predefined object of standard stream in C++. The operator << is called 'insertion' or 'put to' operator. It inserts the content on its right to the object on its left.

```
cout<<a;
```

In the above case, the statement will display the content of the variable 'a'.

The input operator(input using *cin*)

A statement

```
cin>>a;
```

is an input statement. This causes the program to wait for the user to type and give some input. The given input is stored in the variable a.

Here, the identifier 'cin', pronounced as 'see in' is an object of standard input stream. The operator '>>' is called 'extraction' or 'get from' operator. It extracts or gets value from keyboard and assigns it to the variable on its right.

Cascading I/O operators(multiple input/output)

The i/o operators can be used repeatedly in a single i/o statements as follows.

```
cout<<a<<b<<c;
cin>>x>>y>>z;
```

These are perfectly legal. The above cout statement first sends the value of 'a' to cout, then sends the value of b and then sends the value of c. Similarly, the cin statement first reads a value and stores in x, then reads again and stores in y and then in z. The multiple uses of i/o operators in one statement is called cascading.

The iostream header file

The directive '#include<iostream.h>' causes the preprocessor to add the contents of iostream.h file to the program. It contains the declarations of identifiers cout, cin and the operators << and >>. So, the header file iostream should always be included at the beginning if we need to use cin, cout, << and >> operators in our program.

Tokens: Tokens are the smallest individual units in a program. Keywords, identifiers, constants, strings and operators are tokens in C++.

Keywords: Keywords are explicitly reserved identifiers and can not be used as names for the program variables or other user-defined program elements. Some keywords are **int**, **auto**, **switch**, **case**, **do**, **else**, **public**, **default**, **continue** etc.

Functions

A function is a single comprehensive unit that performs a specified task. This specified task is repeated each time the function is called. Functions break large programs into smaller tasks. They increase the modularity of the programs and reduce code redundancy. Like in C, C++ programs also should contain a main function, where the program always begins execution. The main function may call other functions, which in turn will again call other functions.

When a function is called, control is transferred to the first statement of the function body. Once the statements of the function get executed (when the last closing bracket is encountered) the program control return to the place from where this function was called.

Data types in C++

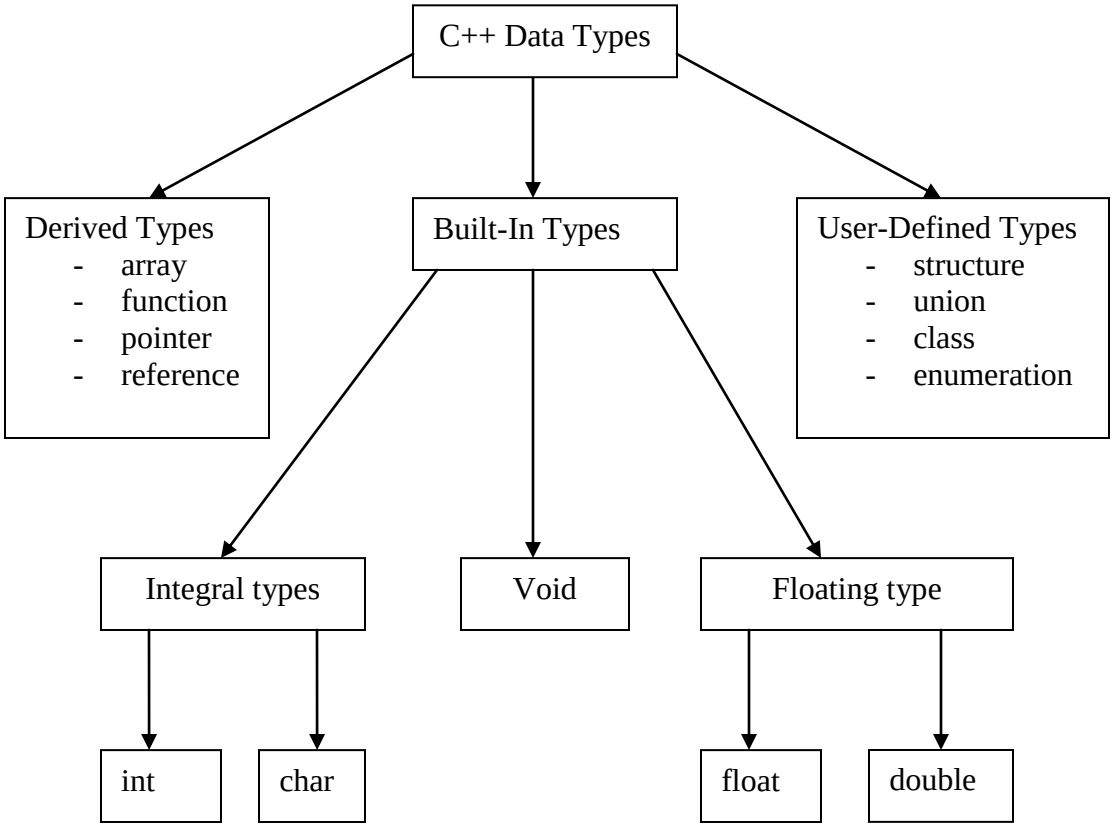


Fig. Hierarchy of C++ Data types

Enumerated Data Types

Like structures, enumerated data type is another user defined data type. Enumerated means that all the values are listed. They are used when we know a finite list of values that a data type can take on or it is an alternative way for creating symbolic constants. The 'enum' keyword automatically lists a list of words and assign them values 0,1,2…

Eg. enum shape {circle, square, triangle};
enum Boolean {true, false};
enum switch {on, off};

The above example is equivalent to

```
const  circle = 0;
const square = 1;
const triangle = 2;
```

We can even use standard arithmetic operator on enum types. We can also use relational operators when suitable. This is because, the enum data types are internally treated as integers.

Once we specify the data type, we need to define variables of that type.

Eg. shape s1,s2;

Now the variables s1, s2 can hold only the members of 'shape' data type (those are circle, square and triangle) and can not hold anything except these values. If other values are given, error will be generated.

An example

```
#include<iostream.h>
#include<conio.h>
enum days {sun, mon, tue, wed, thur, fri, sat};
void main()
{
        days d1,d2;
        d1 = sun;
        d2 = thur;
        int diff = d2-d1;  // using arithmetic operator
        cout<<"Days between"<<diff<<endl;
        if(d1<d2)   // using relational operator
                cout<<"d1 comes first";
        getch();
}
```

Reference variables

Reference variables are new type of variable introduced in C++. It provides an alias (another name) for a previously defined variable. A syntax to create a reference variable is

```
        data-type  &reference-name = variable name;
```

eg-

```
        float total = 100;
        float &sum = total; // creating reference variable for 'total'.
```

In the above example, we are creating a reference variable 'sum' for an existing variable 'total'. Now these can be used interchangeably. Both of these names refer to same data object in memory. If the value is manipulated and changed using one name then it will change for another also. Eg- the statement

total = total + 200; will change value of 'total' to 300. And it will also change for 'sum'. So the statements

```
cout<<sum;
```

cout<<total; both will print 300. This is because both the variables use same data object in memory.

- A reference variable must be initialized at the time of declaration, since this will establish correspondence between the reference and the data object which it names.
- The symbol & is not an address operator here. The notation int & means reference to integer type data.
- References can also be created for user defined data types like structures and classes.

- Another application of reference variable is in passing arguments to function.

In general, arguments are passed by value. The called function creates a new value of the same type as the argument and copies the argument value into it. The function does not access the actual value. Although this provides security to the actual data, it is not suitable if we need to modify actual data. For such situations, we can use reference. Instead of value, a reference to the original variable is passed to the called function. This is called **calling function by reference**. The advantage is that the called function can use actual variable and not its copy only. Likewise, we can also return values using reference. (Example- swapping values)

Manipulators

The manipulators are operators used with insertion operator "<<' to format or manipulate the data display. 'endl' and 'setw' are most common manipulators.

endl manipulator causes a linefeed to be inserted into the output stream. i.e the cursor moves to next line. It is similar to '\n' character.

Eg-

```
cout<<"Kathmandu"<<endl;
cout<<"Nepal";
```

See output

setw manipulator specify a field width to a number or string that follows it and force them to be printed right justified. The field width is given as an argument to this manipulator.

Eg-

```
x = 456; y = 40;
cout<<setw(5)<<x<<setw(5)<<y;
```

The manipulator will specify a field 5 for printing the value of x. The value is right justified within the fields as shown below. For y, it will specify again space of width 5, right justifies and prints.

		4	5	6

x

			4	0

y

Functions

A function is a single comprehensive unit that performs a specified task. This specified task is repeated each time the function is called. Functions break large programs into smaller tasks. They increase the modularity of the programs and reduce code redundancy.

Like in C, C++ programs also should contain a main function, where the program always begins execution. The main function may call other functions, which in turn will again call other functions.
When a function is called, control is transferred to the first statement of the function body. Once the statements of the function get executed (when the last closing bracket is encountered) the program control return to the place from where this function was called.

Function Prototype (Function declaration)
Function prototype lets the compiler know the structure of function in terms of its name, number and type of arguments and its return type.

Syntax:

return-type function-name(datatype1, datatype2, …,datatype n);

Function Call
Function call is the process of making use of function by providing it with the parameters it needs. We call a function as follows.

function-name (argument1, argument2, .. ,argument n);

Function Definition
Function definition is a process of defining how it does what it does or in other words, during function definition, we list the series of codes that carry out the task of the function. A function is defined as follows,

```
return-type function-name(datatype1 variable1, datatype2 var2, …., datatype n var n)
{
        …… ;
        ……; //body of the function
        ……;
}
```

Default Arguments

In C++, a function can be called without specifying all its arguments. But it does not work on any general function. The function declaration must provide default values for those arguments that are not specified. When the arguments are missing from function call, default value will be used for calculation.

```
#include<iostream.h>
float interest(int p, int t = 5, float r = 5.0);
main()
{
        float rate, i1,i2,i3;
        int pr , yr;
        cout<<"Enter principal, rate and year";
        cin>>pr>>rate>>yr;
```

```
        i1=interest(pr ,yr ,rate);
        i2=interest(pr , yr);
        i3=interest(pr);
        cout<<i1<<i2<<i3;
        return(0);
}

float interest(int p, int t, float r)
{
        return((p*t*r)/100);
}
```

In the above program, t and r has default arguments. If we give, as input, values for pr, rate and yr as 5000, 10 and 2, the output will be

1000 500 1250

NOTE: The default arguments are specified in function declaration only and not in function definition.
Only the trailing arguments can have default values. We must add defaults from right to left. We cannot provide a default value to a particular argument at the middle of an argument list. Default arguments are used in the situation where some arguments have same value. For eg., interest rate in a bank remains same for all customers for certain time.

Inline Functions

We say that using function s in a program is to save some memory space because all the calls to the functions cause the same code to be executed. However, every time a function is called, it takes a lot of extra time in executing a series of instructions. Since the tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function are carried out when a function is called. When a function is small, considerable amount of time is spent in such overheads.

C++ has a solution for this problem. To eliminate the cost of calls to small functions, C++ proposed a new feature called INLINE function.
When a function is defined as inline, compiler copies it s body where the function call is made, instead of transferring control to that function.
A function is made inline by using a keyword “inline” before the function definition.

Eg.

```
inline void calculate_area(int l,int b)
{
        return(l * b);
}
```

It should be noted that, the inline keyword merely sends request, not a command, to a compiler. The compiler may not always accept this request. Some situations where inline expansion may not work are

- for functions having loop, switch or goto statements
- for recursive functions
- functions with static variables
- for functions not returning values, if a return statement exists

Inline functions must be defined before they are called.
Eg.

```
#include<iostream.h>

inline float lbtokg(float lbs)
{
        return (0.453 * lbs);
}


main()
{
        float lbs, kgs;
        cout<<"Enter weight in lbs:";
        cin>>lbs;
        kgs=lbtokg(lbs);
        cout<<"Weight in kg is "<<kgs;
        return (0);
}
```

Exercise:

When do we use inline function? Explain with example.
When do we use default argument? Explain with example.

Function Overloading

Function that share the same name are said to be **overloaded functions** and the process is referred to as **function overloading**. i.e. function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters, or different sequence of parameters or different number of parameters. The number, type or sequence of parameters for a function is called the

function signature. When we call the function, appropriate function is called based on the parameter passed. Two functions differing only in their return type can not be overloaded. For eg-
int add(int , int) and float add(int, int)

A function call first matches the declaration having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection will involve the following steps:

- the compiler first tries to find an exact match in which the types of actual arguments are the same and uses that function
- if an exact match is not found, the compiler uses the integral promotion to the actual parameters, such as,
    - char to int
    - float to double to find the match
- If both of the above fail, the compiler tries to use the built-in conversions and then uses the function whose match is unique.

```
#include<iostream.h>
//function declaration
float perimeter(float);
int perimeter(int,int);
int perimeter(int,int,int);

main()
{
        cout<<"Perimeter of a circle: "<<perimeter(2.0)<<endl;
        cout<<"Perimeter of a rectangle: "<<perimeter(10,10)<<endl;
        cout<<"Perimeter of a triangle: "<<perimeter(5,10,15);
        return (0);
}

//function definition
float perimeter(float r)
{
        return(2*3.14*r);
}
int perimeter(int l,int b)
{
        return(2*(l+b));
}
int perimeter(int a,int b,int c)
{
        return(a+b+c);
}
```

In the above program, a function "perimeter" has been overloaded. The output will be as follows:

Perimeter of a circle 12.56
Perimeter of a rectangle 40
Perimeter of a triangle 30

Recursive function

Recursion is a powerful technique of writing complex algoriths in an easy way. It defines the problem in terms of itself. In this technique , a large problem is divided into smaller problem of similar nature as original problem.so that the smaller problem is easy to solve and in the most case they can be solved easily. Hence to implement this techinique, a programming language support the function that is capable of calling itself. C++ support such function and these function are called recursive functions

For example: to find the factorial of a given number

```
Int mani()
{
        int num;
        cout<<"Enter a number";
        cin>>num;
        int f=fact(num);
        cout>>"the factorial of given number is"<<f;
        getch();
        return 0;

}

int fact(int num)
{
        If(num==0)
                Return 1;
        Else
                Return (num*fact(num-1));
}
```

Classes and Objects

One of the unique facilities provided by C language is "structure". Structures were used to group logically related data items. It was termed as a user defined data-type. Once the structure type was defined, we can create variables of that type.

Eg.

```
                struct student
                {
```

```
        char name[15];
        int roll_no;
        float total_marks;
};
```

For the above structure, we can define variables like

```
struct student s1,s2,s3;
```

But there are certain limitations in C structures.

- They don't allow data hiding. The structure members can be accessed using structure variable by any function anywhere in the scope.
- C does not allow the structure data type to be treated like built in data type.

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main precepts of OOP.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as private so that they can not be accessed directly by the external functions. In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword **struct** can be omitted in the declaration of structure variables.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structure for holding only data, and classes to hold both the data and functions.

The only difference between a structure and a class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.

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 data-type. So, classes are user-defined data types and behave like the built-in types of a programming language.
Classes allow the data and functions to be hidden, if necessary, from external use. Class, being a user-defined data type, we can create any number of variables for that class. These variables are called **Objects**.

Specifying a class

Specification of a class consists of two parts.

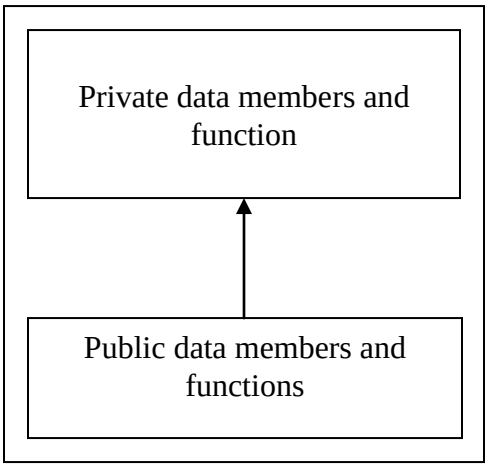
- class declaration
- function definition

Syntax:

```
        class class-name
        {
```

```
private:
        Variable declarations
        Function declarations
public:
        Variable declarations
        Function declarations

};
```

- the class specification starts with a keyword "class" (like "struct" for structures), followed by a class-name. The class-name is an identifier.
- The body of the class is enclosed within braces and terminated by a semicolon.
- The functions and variables are collectively called class-members. The variables are, specially, called data members while the functions are called member functions.
- The two new keywords inside the above specification are – private and public. Those keywords are termed as **access-specifiers**, they are also termed as **visibility labels**. These are followed by colons.
    - The class members that have been declared as private can be accessed only from within the class. i.e., only the member functions can have access to the private data members and private functions.
    - On the other hand, public members can be accessed from anywhere outside the class (using object and dot operator).



    - All the class members are private by default. So, the keyword "private" is optional.
    - If both the labels are missing, then, by default, all the members will be private and the class will be completely inaccessible by the outsiders (hence it won't be useful at all).

The key feature of OOP is data hiding. Generally, data within a class is made private and the functions are public. So, data will be safe from accidental manipulations, while the functions can be accessed from outside the class. However, it is not always necessary that

the data must be private and functions public. In some cases, data may be public too and functions may be private.

Example of a class

```
class Test
{       int x,y;
        public:
        void get_data()
        {
            cin>>x>>y;
        }
        void put_data()
        {
           cout<<x<<y;
        }
        };
```

Creating Objects

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

```
Test t1; //memory for t1 is allocated.
```

The above statement creates a variable t1 of type Test. The class variables are known as **objects.** So, t1 is called object of class Test. We may also declare more than one object as follows.

```
Test t1, t2, t3;
```

When object is created, the necessary memory space is allocated to this object. The specification of class does not create any memory space for the objects. Objects can be created when a class is defined, as follows:

```
class Employee
{
        int id;
        char name[20];
public:
        void getname();
        void putname();
}e1,e2,e3;
```

The **object**s are also called **instance**s of the class. In terms of object, a class can be defined as a collection of objects (or instances) of similar type.

Accessing class members

The class members are accessed using a **dot operator**. But this works only for the public members. The dot operator is called **member access operator**.

The general format is

```
class-object.class-member ;
```

For public data members (if any), we can use following syntax to access them.

class-object.data-member ;

eg. obj1.data1 = obj1.data2 + obj1.data3;

In the above example, obj1 is an object of class. data1, data2, data3 are its public members.

For public functions, we can use following format to call them

class-object.function-name(argument-list);

eg. e1.getdata();

In the above example, e1 is an object of a class and getdata() is its member function. Here, no arguments have been listed since the function does not take any argument.

The private data and functions of a class can be accessed only through the member functions of that class.

Eg.

```
class A
{
        int x,y;
        void fu1();
public:
        int z;
        void fu2();
};

-   - - - - - - - - - -
-   - - - - - - - - - - -
A obj1;
obj1.x = 0; //generates error since x is private and can be accessed only thro' member functions
obj1.z = 0; //valid
obj1.fu1(); //generates error since fu1() is private
obj1.fu2();  //valid
```

Defining member functions (i.e. writing the body of a 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. (As has been done in the previous example of class Employee).

General form:

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

The function is, generally, defined immediately after the class-specifier. The function-name is proceeded by

- return-type of the function
- class-name to which the function belongs
- symbol with double colons(::). This symbol is called the scope resolution operator. This operator tells the compiler that the function belongs to the class class-name.

Eg.

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

In this example, the return-type is void which means the function "getdata()" doesn't return any value. The scope-resolution operator tells that the function "getdata()" is a member of the class "Employee". The argument list is also empty.

The member functions have some special characteristics:

- A program may have several different classes. These classes can use same function name. The scope-resolution operator will resolve which belongs to whom.
- Member functions can directly access private data of that class. A non-member function cannot do so. (Exception is friend function)
- A member function can call another member function directly, without using the dot operator.

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.

Eg.

```
class A
{
        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.

Eg.

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

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

Function definition with the 'inline' qualifier/keyword. This qualifier will make the function 'getdata()' an inline function

Nested Member Functions

An object of the class using dot operator generally, calls a member function of a class. 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".

Eg.

```
#include<iostream.h>

class Addition
{
        int a,b,sum;
        public:
        void read();
        void show();
        int add();
};
void Addition::read()
{
        cout<<"Enter a and b";
        cin>>a>>b;
}
void Addition::add()
{
        return(a+b);
```

```
}
void Addition::show()
{
        sum=add();    // nesting of function i.e member function called from another
member function.
        cout<<endl<<"Sum of a and b is: "<<sum;
}

main()
{
        Addition a1;
        a1.read( );
        a1.show( );
        return(0);
}
```

The output of this program will be as follows:

```
        Enter a and b: 2
        3
        Sum of a and b is:5
```

Private Member Functions

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 it using dot operator.

Eg.

```
        class A
        {
                int a,b;
                void read(); //private member function

                public:
                void update();
                void write();
        };
        void A::update()
        {
                read(); //called from update() function. No object used.
        }

        - - - - - - - - - - - - -
        - - - - - - - - - - - - - -
```

If a1 is an object of A, then the following statement is not valid.

```
         a1.read();
```

This is because, read() is a private member function, hence cant be called using object and dot operator.

Memory Allocation for Objects

- As in the case of structures, memory space for objects is allocated when they are declared and not when the class is specified.
- But, we have to note one thing. The member functions are created and placed in the memory only once, when they are defined as a part of a class specification. All the objects belonging to the particular class will use the same member functions; no separate space is allocated for member functions when objects are created.
- However, the data members will hold different values for different object. So, space for member data is allocated separately for each object.

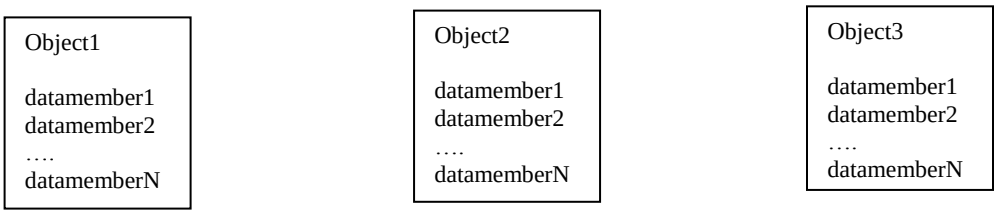


fig: Objects in memory

Array of Objects

Array can be created of any data-type. Since a class is also a user defined data type, we can create array of objects (the variable of type class). Such an array is called an **array of objects**.

For eg. we had specified a class called Employee in some example above. If we have to keep records of 20 employees in [illegible]ts, then instead of creating 20 separate variables, w[illegible].

Employee dept1
Employee dept2

The arrays dept1 and dept2 [illegible] objects of type Employee).

Since the array of objects is an array, we can access the members of the classes as follows:

dept1[0].getdata();
dept2[3].getdata();

Member function 1
Member function 2
.
.
member function N

Objects as function arguments

Like any other variable, objects can also be passed to the function, as an argument. There are two ways of doing this.

- pass by value
- pass by reference

In the "pass by value", the copy of the object is passed to the function. So, the changes made to the object inside the function do not affect the actual object. On the other hand, address of the object is passed in the case of pass by reference. So, changes made to the object inside the function are reflected in the actual object. The second method is considered more efficient.

Eg.

```
class Height
{
        int feet;
        int inches;
        public:
                void getHeight()
                {
                cout<<"Enter height in feet and inches";
                cin>>feet>>inches;
                }

                void putHeight()
                {
                cout<<"Feet:"<<feet;
                cout<<" and Inches:"<<inches;
                }

                void sum(Height,Height);
};

        void Height::sum(Height h1, Height h2)
        {
                inches = h1.inches + h2.inches;
                feet=inches/12;
                inches=inches%12;
                feet=feet + h1.feet + h2.feet;
        }

        int main()
        {
                Height h1,h2,h3;
                h1.getHeight();
                h2.getHeight();
                h3.sum(h1,h2);
```

```
        cout<<"Height 1:"<<h1.putHeight();
        cout<<"Height 2:"<<h2.putHeight();
        cout<<"Height 3:"<<h3.putHeight();
        return(0);
}
```

In the above program, the function sum() takes two objects as arguments.

NOTE: A function can, not only, take objects as arguments, but they can also return object.

Eg.

In the above program if we modify the function sum() as follows:

```
Height Height::sum(Height h1,Height h2) //function with return type Height
{
        Height h3;
        h3.inches = h1.inches + h2.inches;
         h3.feet= h3.inches / 12;
        h3.inches= h3.inches % 12;
        h3.feet = h3.feet + h1.feet + h2.feet;
        return (h3); //object of type height returned
}
```

The above function will return an object h3 at the end of its execution.

Friend Function

A function is said to be a friend function of a class if it can access the members (including private members) of the class even if it is not the member function of this class.

In other words, a friend function is a non-member function that has access to the private members of the class.

Characteristics of a friend function:

- A friend function can be either global or a member of some other class.
- Since, friend function is not a part of the class, it can be declared anywhere in the public, private and protected section of the class.
- It cannot be called by using the object of the class since it is not in the scope of the class.
- It is called like a normal function without the help of any object.
- Unlike member functions, it cannot access member names directly. So, it has to use an object name and dot operator with each member name (like A.x)
- It, generally, takes objects as arguments.

A friend function is declared as follows:

```
class A
{
-- ---------- ----- 
-- ---------- ----- 
public:
        ----------
        ----------
friend void abc(void);                  //friend function declaration
};
```

The declaration is preceded by the keyword "friend". Its definition is written somewhere outside the class.
Eg.

```
class Avg
{
        int n1,n2;
        public:
         void getn()
        {
                cin>>n1>>n2;
        }
        friend int average(Avg a);
};
        //here "friend" is a keyword to specify that the function declared is a friend function
        //int is a return type of the friend function
        //average is the name of the friend function
        //Avg is the type of the argument type
        //a is an argument

int average(Avg a) //friend function definition
{
        return((a.n1 + a.n2)/2);
}

int main()
{
        Avg obj;
        obj.getn();
        cout<<"Mean:"<<average(obj);        //friend function called
        return(0);
}
```

Suppose we want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private

data. In this situation there is nothing like a friend function. Here is an example that shows how friend functions can act as a bridge between two classes. Read program carefully!!!!!!!!

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

class beta;   // needed for frenfunction declaration

class alpha
{
        private:
        int data;
        public:
        void get_data()
        {
                cin>>data;
        }
        friend int frenfunction(alpha, beta);  // friend function
};

class beta
{
        private:
        int data;
        public:
        void get_data()
        {
                cin>>data;
        }
        friend int frenfunction(alpha, beta);  // friend function
};

int frenfunction(alpha a,beta b)
{
        return(a.data+b.data);
}
main()
{
        alpha aa;
        beta bb;
        aa.get_data();
        bb.get_data();
        cout<<frenfunction(aa,bb)<<endl;
        getch();
        return 0;
```

}

Static Data Members

We have known that, each object of a class maintain their own copy of member data. In some cases, it may be necessary that all objects of a class have access to the same copy of a single variable. This can be made possible by using static variables.

- Only one copy of the static member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.
- It is initialized to zero when the first object of its class is created.
- Also known as class variable.

A static variable is declared using the "static" keyword.

```
class A
{
        static int count;
        int variable;
        public:
        A()
        {
                count++;
        }
        void get_var()
        {
                cin>>variable;
        }
        void put_var()
        {
                cout<<variable
        }
        void put_count()
        {
                cout<<count;
        }
};
int A::count;

main()
{
        A a,b,c;
        a.put_count();
        b.put_count();
        c.put_count();
        return(0);
}
```

The type and scope of each static member variable is defined outside the class definition. It is necessary since static data member are stored separately, rather than as a part of the object.

Also note, static variable can be assigned some initial value.
Eg.

```
int A::count=10;
```

will assign count the initial value 10

Output:

1 2 3

Static Member Function

In a class, functions can also be declared as static. Properties of static functions are

- they can access only other STATIC members (functions or variables) declared in the same class
- they can be called using class name
  eg. class_name::function_name

Example

```
class A
{
        int no;
        static int count;          //static member
        public:
        void set_no()
        {
                count++;
                no = count;
        }
        void put_no()
        {
                cout<<"No. is :"<<no;
        }
        static void put_count()          //static function accessing static member
        {
                cout<<endl<<"count:"<<count;
        }
};
int A::count;
main()
{
        A a1,a2;
        a1.set_no();
        a2.set_no();
        A::put_count();
        a1.set_no();
        a2.set_no();
        A::put_count();
        a1.put_no();
        a2.put_no();
        return(0);
```

}

Check output of this program.

Constructors and Destructors

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **input()** and **output()** to provide initial values to the private member variables. For example, the statement

t.input();

invokes the member function input(), which assigns the initial values to the data items of object **t**. Similarly, the statement

t.input(100,200);

passes the initial values as arguments to the function input(), where these values are assigned to the private variables of object **t**. All these function call statements are used with the appropriate objects that have already been created. These functions can not be used to initialize the member variables at the time of creation of their objects.

One of the aims of C++ is to create user-defined data types such as class that behave very similar to the built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

int x = 10;
float y = 56.67;

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. There are some features of class that enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the ***constructor*** which enables an object to initialize itself when it is created. This is known as ***automatic initialization*** of objects.

It also provides another member function called the ***destructor*** that destroys the objects when they are no longer required.

Constructors

A constructor is a special member function whose task is to initialize the objects of its class. It has the name same as that of class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

- They are also used to allocate memory for a class object.
- They execute automatically when an object of a class is created.
- Constructor's name is same as that of class name.
- They should be declared in the "public" section.
- They do not have return types, not even void and therefore, and they cannot return any values.
- Like C++ functions, they can have default arguments.
- Constructor is NOT called when a pointer of a class is created.

There are, basically, three types of constructors.

- Default constructors
- Parameterized Constructors
- Copy Constructors

Default Constructors

A constructor that does not take any parameter is called default constructor. There are three possible situations for this.

1. If we do not provide any constructor with a class, the compiler provided one would be the default constructor. And it does not do anything other than allocating memory for the class object.

```
class A
{
    //no constructor
};
```

2. If we provide a constructor without any arguments then that is the default constructor.

```
class A
{
    A()
    {}
    //or
    A(void)
    {}

};
```

3. If we provide constructor with all default arguments, then that can also be considered as the default constructor.

```
class A
{
        A (int x=5)
        {}
};
```

Parameterized Constructor

A constructor that takes arguments is called a parameterized constructor. Arguments are passed when the objects are created. This can be done in two ways.

- by calling the constructor explicitly
- by calling the constructor implicitly

For eg.

```
class A
{
                                int m,n;
                                public:
                                        A(int x, int y); //parameterized constructor
                                                                {
                                 m=x;
        n=y;
        }
                                  -                                -       -
                                  -                                -       -

};

main()
{
                                A obj1(10,20); //implicit call
                                A obj2 =A(10,20); //explicit call
                                  -                                -       -
                                  -                                -       -
                                  -                                -       -
                                  -                                -       -
}
```

Class name

Object name

Constructor call with parameter

NOTE: Constructor functions can also be defined explicitly using scope resolution operator.

Copy Constructor

A copy constructor is called when an object is created by copying an existing object.

Eg.

A obj2(obj1);

The above statement would define the object obj2 and initialize it to the values of obj1.
The above statement can also be written as,

A obj2=obj1;

The process of initializing through a copy constructor is known as ***copy initialization***.
A copy constructor takes a reference to an object of the same class as itself (as an argument). ***We cannot pass the argument by value to a copy constructor***. When no copy constructor is defined, the compiler supplies its own copy constructor. Remember the statement

obj2 = obj1;

will not invoke the copy constructor. However, if obj1 and obj2 are objects, this statement is legal and simply assigns the values of obj1 to obj2, member by member. This is the task of the overloaded assignment operator (=).

Eg.

```
class Data
{
                int info;
                public:
                                Data()
        {}
                        Data(int a)         // parameterized constructor
                                {
                                        info
        }
        Data(Data &x)   // copy constructor
        {
                        info=x.info;
        }
        void display()
        {
                        cout<<info;
        }
};

int main()
{
                Data d1(5); //parameterized constructor is called.
                Data d2(d1); //copy constructor is called here.
```

```
Data d3=d2; //again a copy constructor is called here.
Data d4;
d4 = d1;     // copy constructor not called
cout<<"An info stored in d1";
d1.display();
cout<<"An info stored in d2";
d2.display();

cout<<"An info stored in d3";
d3.display();
cout<<"An info stored in d4";
d4.display();

return(0);
}
```

Constructors are called thrice in this program. The first one d1 calls the parameterized constructor, while the other two calls the copy constructor

The output will be

```
An info stored in d1 5
An info stored in d2 5
An info stored in d3 5
An info stored in d4 5
```

Constructor Overloading

The process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor is defined in a class, it is called constructor overloading.

In the above example of class Data, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply one argument, while the third one gets invoked when an object is passed as an argument.

Eg.

```
Data obj1;
```

This statement would automatically invoke the first one.

```
Data obj2(5); invokes 2nd constructor
Data obj3(obj2); invokes 3rd constructor
```

Constructor with default argument

Like functions, constructors can also have default arguments.

Eg.

```
class A
{
int a,b,c;
 pulic:
A()
 {
```

a=0

```
        }
        A(int x,int y=10,int z=20)
        {
                a=x;b=y;c=z;
        }
        void display()
        {
                cout<<a<<b<<c;
        }
};

main()
{
                A obj1;
        A obj2(5,10,15);
                A obj3(6);
        obj1.display();
        obj2.display();
        obj3.display();
                return(0);
}
```

Output:

```
0 0 0
5 10 15
6 10 20
```

In this program, there are two constructors. The second one has default parameters. Such a constructor is called a constructor with default argument. These constructors have all the properties of the **functions with default argument.**

It is important to distinguish between the default constructor A::A() and the default argument constructor A::A (int x = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A   a;
```

The ambiguity is whether to call A::A() or A::A(int x = 0).

Example

```
#include<iostream.h>
#include<conio.h>
class test
{
```

```
  int x,y;
  public:
  //test()   // No need to use this default constructor
  //{}
  test(int p=10,int q = 20) // when this constructor is called with no arguments, it
  {                         // becomes default constructor
    x = p;
    y = q;
  }
  void display()
  {
     cout<<"X = "<<x<<"\n"<<"Y = "<<y<<endl;
  }
};

void main()
{
  int x,y;
  cin>>x>>y;
  test t(x,y);  // call to default argument constructor
  t.display();
  test t1;       //  No argument is passed, so becomes default constructor
  t1.display();
  getch();
}
```

Dynamic initialization of objects

Class objects can be initialized dynamically (i.e. at the run time). The users provide the values at the run time.

Advantage: various initialization formats can be provided using constructor overloading.

Eg.

```
class Area
{
int l,b,area;
  public:
        Area()
        {
                l=1;
                b=1;
                area=1;
        }
        Area(int a, int br)
        {
          l=a;
          b=br;
        }
        void main()
        {
          Area a1,a2;
          int l,b,len;
          cin>>l>>b;
          a1=Area(l,b);
          cin>>len;
          a2=Area(len);
          - - - - - - - - -
          - - -- - - - - - -
        }
```

NOTE: In dynamic initialization, we are explicitly calling the required constructor. And we haven't used the object and dot operator.

```
                                        area=l*b;
        }
        Area(int a)
        {
                                          l=a;
                                         b=0;
                                       area=l*l;
        }
};
```

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 (~).

Eg.

A destructor for the class Area will look like

```
                                                    ~Area()
                                                  { - - - - - - -
    - - - - - -
        }
```

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.

```
#include<iostream.h>
#include<conio.h>
class A
{
    static int count;
    public:
    A()
    {
        count++;
        cout<<count<<endl;
    }
    ~A()
    {
        cout<<count<<endl;
        count--;
    }
};

int A::count;

main()
{
```

```
        A a1,a2,a3;
        getch();
        return 0;
}
```

Operator Overloading

The concept of overloading can be applied to operators as well. Operator overloading is the mechanism of giving special meanings to an operator. It provides a flexible option for the operations of new definitions for most of the C++ operators. In other words, operator overloading refers to giving the normal C++ operators (such as +, *, <=, += etc) additional meanings when they are applied to user-defined data types. In general,

a = b + c; works only with basic types like 'int' and 'float', and attempting to apply it when a, b and c are objects of a user defined class will cause complaints from the compiler. But, using overloading, we can make this statement legal even when a, b and c are user defined types (objects).

There are two types of operator overloading

- Unary operator overloading and
- Binary operator overloading

Unary operator overloading

Unary operators are those operators that act on a single operand. ++, -- are unary operators. The following program overloads the ++ unary operator for the distance class to increment the data number by one.

```
class Distance
{
        int feet;
        float inch;
        public:
        Distance (int f, float i);
        void operator++(void);
        void display();
};
Distance :: Distance (int f, float i)
{
        feet = f ; inch = i;
}
void Distance :: display()
{
        cout<<"Distance in feet"<<feet<<endl;
        cout<<"Distance in inch"<<inch<<endl;
```

```
}
void Distance :: operator ++(void)
{
        feet++;
        inch++;
}
void main()
{
        Distance dist(10,10);
        ++dist;
        dist.display();
}
```

In the above example, the ++ unary operator has been overloaded in the function void Distance :: operator ++(void). In this overloaded function, data members feet and inch are increased by one. This function is called at the second line '++dist' in the main.

General syntax for defining operator overloading

```
return-type   classname :: operator  operator-to-overload (arg. list)
{
        //func body
}
```

The keyword '**operator**' is used to overload an operator. This declaration tells the compiler to call this member function whenever the ++ operator is encountered, provided the operands are of user-defined type.

//Another Example - overloading unary minus operator

```
class abc
{
        int x,y,z;
        public:
        void getdata(int a,int b,int c)
        {
                x = a;
                y = b;
                z = c;
        }
        void display()
        {
                cout<<x<<y<<z<<endl;
        }
        void operator -();
};
void abc::operator-()
{
        x = -x;
```

```
        y = -y;
        z = -z;
}

main()
{
        abc a;
        a.getdata(4,-5,6);
        a.display();
        -a;
        a.display();
        getch();
        return 0;
}
```

Operator Return Values

The operator++() function can return a value. If we use a statement like this

```
c1 = ++c2;
```

For this we have to define the ++ operator to have a return type object of a class in the operator++ function. That is the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1. the example given below illustrates this.

```
class Counter

{
        int count;
        public:
        Counter()
        {
                count = 0;
        }
        Counter(int c)
        {
                count = c;
        }

        Counter operator++()
        {
                return Counter(++count);
        }
        void put_count()
        {
                cout<<count<<endl;
```

```
        }
};
main()
{
        Counter c1,c2;
        c1.put_count();
        c2.put_count();
        c2 = ++c1;
        c1.put_count();
        c2.put_count();
        getch();
        return 0;
}
```

Postfix Notation

We have overloaded the ++ operator in prefix form. What about postfix, where the variable is incremented after its value is used in the expression?

c1++

to make both versions of the increment operator work, we define two overloaded ++ operators as follows

```
class Counter

{
        int count;
        public:
        Counter()
        {
                count = 0;
        }
        Counter(int c)
        {
                count = c;
        }

        Counter operator++()
        {
                return Counter(++count);
        }
        Counter operator++(int)
        {
                return Counter(count++);
         }
        void put_count()
```

```
        {
                cout<<count<<endl;
        }
};
main()
{
        Counter c1,c2,c3;
        c1.put_count();
        c2.put_count();
        c2 = ++c1;
        c3 = c2++;
        c1.put_count();
        c2.put_count();
        c3.put_count();
        getch();
        return 0;
}
```

Now there are two different declarators for overloading the ++ operator. The one, for prefix notation, is

Counter operator ++()

The one, for postfix notation, is

Counter operator ++(int)

The only difference is the **int** in the parentheses. This **int** is not really an argument, and it does not mean integer. It's simply a signal to the compiler to create the postfix version of the operator. The designers of C++ are fond of recycling existing operators and keywords to play multiple roles, and **int** is the one they chose to indicate postfix.

Overloading binary operators

Binary operators are those that work on two operands. Examples are +,-,*, /, % for arithmetic operations, +=,-=,*= and /= for assignment operations and >, <, <=,>=, == and! = for comparison operations.

Overloading a binary operator is similar to overloading unary operator except that a binary operator requires an additional parameter. The following code fragment overloads binary + operator. It adds two objects of type 'distance'.

```
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
```

```
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance operator + (Distance);
};
Distance Distance :: operator + (Distance d2)
{
        int m = meter + d2.meter;
        int cm = centimeter + d2.centimeter;
        if(cm >= 100)
        {
                cm -= 100;
                m++;
        }
        return Distance (m,cm);
}
main()
{
        Distance d1(4,50);
        Distance d2,d3,d4;
        d2.getDist();
        d3 = d1 + d2;  // Invokes operator+() function
        d4 = d3.operator+(d1); // usual function call syntax
        d3.show();
        d4.show();
        return 0;
}
```

NOTE Here the function operator +(Distance) does not need two arguments, since there are two objects to be added. The argument on the left side the operator (d1 here) is the

object of which the operator is a member. The object on the right side of the operator (d2 here) must be furnished as an argument to the operator.

Overloading Binary Operators Using Friend Function

Friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one. The distance addition program discussed above can be modified using friend function as follows:

```
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        friend Distance operator + (Distance,Distance);
};
Distance operator + (Distance d1,Distance d2)
{
        int m = d1.meter + d2.meter;
        int cm = d1.centimeter + d2.centimeter;
        if(cm >= 100)
        {
                cm -= 100;
                m++;
```

```
        }
        return Distance (m,cm);
}
main()
{
        Distance d1(4,50);
        Distance d2,d3;
        d2.getDist();
        d3 = d1 + d2;
        d3.show();
        getch();
        return 0;
}
```

---------- Examples Programs----------

//Overloading == operator

```
class equal
{
        int feet;
        float inch;
        public:
        equal()
        {}
        equal(int f,float i)
        {
                feet = f;
                inch = i;
        }
        void display()
        {
                cout<<feet<<inch;
        }
        int operator ==(equal);
};
int equal::operator==(equal e)
{
    if(feet == e.feet && inch == e.inch)
    return 1;
    else
    return 0;
}

main()
{
        equal e1(6,4.4);
        equal e2(6,4.2);
```

```
        if(e1==e2)
        cout<<"Both are of same length";
        else
        cout<<"Not same";
        getch();
        return 0;
}
```
//Overloading == operator to compare two strings
```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 20
class string
{
        private:
        char str[SZ];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char s[])
        {
                strcpy(str,s);
        }
        void getstring()
        {
                cout<<"\nEnter a string ";
                cin>>str;
        }
        int operator ==(string ss)
        {
                return(strcmp(str,ss.str) == 0)?1:0;
        }
        void display()
        {
                cout<<str<<endl;
        }
};

main()
{
        clrscr();
        string s1 = "Nepal";
        string s2 = "Kathmandu";
        string s3;
```

```
        s3.getstring();
        if(s3 == s1)
        cout<<"\nYou typed Nepal";
        else if(s3 == s2)
        cout<<"\nYou typed Kathmandu";
        else
        cout<<"\nNot of both";
        getch();
        return 0;
}
```

//Overloading + operator to concatenate two strings

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 20
class string
{
        private:
        char str[SZ];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char s[])
        {
                strcpy(str,s);
        }
        void display()
        {
                cout<<str<<endl;
        }
        string operator +(string ss)
        {
                string temp;
                if(strlen(str)+strlen(ss.str) < SZ)
                {
                        strcpy(temp.str,str);
                        strcat(temp.str,ss.str);
                }
                else
                {
                        cout<<"\nstring overflow";
                }
                return temp;
```

```
        }
};

main()
{
        string s1 = "Shiva";
        string s2 = "Parbati";
        string s3;
        s1.display();
        s2.display();
        s3 = s1 + s2;
        s3.display();
        cout<<endl;
        getch();
        return 0;
}

//Overloading < operator
class line
{
        int feet;
        float inch;
        public:
        line()
        {}
        line(int f,float i)
        {
                feet = f;
                inch = i;
        }
        void display()
        {
                cout<<feet<<inch;
        }
        int operator<(line);
};
int line::operator<(line l)
{
        float l1 = feet + inch/12;
        float l2 = l.feet + l.inch/12;
        if(l1<l2)
                return 1;
        else
        return 0;
}
```

```
main()
{

        line l1(2,5.5);
        line l2(1,6.3);
        if(l1<l2)
                l1.display();
        else
                l2.display();
        getch();
        return 0;
}
```

Multiple Overloading

We have seen several different uses of + operator: - to add distance and to concatenate strings. We can put both these classes together in the same program, and C++ still knows how to interpret the + operator. It selects the correct function to carry out the addition based on the type of operand. Such an example is given below.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 40
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
```

```
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance operator + (Distance);
};
Distance Distance :: operator + (Distance d2)
{
        int m = meter + d2.meter;
        int cm = centimeter + d2.centimeter;
        if(cm >= 100)
        {
                cm -= 100;
                m++;
        }
        return Distance (m,cm);
}
class string
{
        private:
        char str[SZ];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char s[])
        {
                strcpy(str,s);
        }
        void display()
        {
                cout<<str<<endl;
        }
        string operator +(string ss)
        {
                string temp;
                if(strlen(str)+strlen(ss.str) < SZ)
                {
                        strcpy(temp.str,str);
                        strcat(temp.str,ss.str);
                }
                else
                {
                        cout<<"\nstring overflow";
                }
                return temp;
```

```
        }
};

main()
{

        Distance d1(4,50);
        Distance d2,d3;
        d2.getDist();
        d3 = d1 + d2;
        d3.show();
        string s1 = "Shiva";
        string s2 = "Parbati";
        string s3;
        s1.display();
        s2.display();
        s3 = s1 + s2;
        s3.display();
        getch();
        return 0;
}
```

General rules for overloading operators

- There are some restrictions and limitations to be kept in mind while overloading operators. They are as follows:
- Only the existing operators can be overloaded. New operators can not be created.
- The overloaded operators must have at least one user-defined operand.
- It is not recommended to change the basic meaning of an operator. That is, the plus (+) operator should not be redefined to subtract one value from another.
- Overloaded operators follow syntax rules of the original operators. That can not be overridden.
- Friend functions can not be used to overload certain operators like =, ( ), [ ] and ->.
- Unary operators, overloaded by means of a member functions take no explicit arguments and return no explicit values. But those overloaded by a friend functions take one reference argument.
- Binary operator overloaded through a member function take one explicit argument and those that are overloaded through a friend function take two arguments.
- Binary operators such as +, -, *, and / must explicitly return a value.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

Operator Overloading Restriction

- Operator functions can not have default arguments.
- The operators that can not be overloaded are **.** , **::**, **.***, ?:, sizeof.

- One can not alter the precedence of operators
- One can not change the number of operands that an operator takes.

Type Conversion (Data Conversion)

We use the assignment operator (=) to assign value of one variable to another. For example

x = y;

Where x and y are integer variables. We have also noticed that = assigns the value of one user defined object to another, provided that they are of the same type. For example,

d2 = d1;

Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler does not need any special instructions to use = for the assignment of user-defined objects such as distance objects.

The assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign.

But if the variables on different sides of the = are of different types, then the type of variable on the right side of = needs to be converted to the type of left side variable before the assignment takes place.

Type conversion is the conversion of one data type to another data type.

Conversion Between Basic Types

Consider the statement,

intvar = floatvar;

where intvar is of type int and floatvar is of type float. Here the compiler will call a special routine to convert the value of floatvar, which is expressed in floating point format, to an integer format so that it can be assigned to intvar. There are many such

conversions: from **float** to **double**, **char** to **float** and so on. Each such conversion has its own routine, built into the compiler and called up when the data types on different sides of the = sign so dictate. Such conversions are **implicit conversion**.

Sometimes we want to force the compiler to convert one type to another. For example,

```
int total = 400;
float avg;
avg = float(total) / 5;  // converts value of total to float before division takes place.
```

Conversion Between Objects and Basic Types

When we want to convert between user-defined data types and basic types, we can not rely on built-in conversion routines, since the compiler does not know anything about user-defined types besides what we tell it. Instead, we must write these routines ourselves.

From Basic to Class type Conversion

To go from a basic type to a user defined type, we use constructor. These are sometimes called ***conversion constructors.*** Example-

```
class time
{
        int hrs;
        int min;
        public:
        time()
        {}
        time(int t)
        {
                hrs = t/60;
                min =  t%60;
        }
        void display()
        {
                cout<<"Hours = "<<hrs<<"Minutes = "<<min;
        }
};

void main()
{
  time  t1 = 95; // uses one-argument constructor to convert integer to time
  t1.display();
  getch();

}
```

From User-Defined to Basic Type

When class type data is converted into basic type data, it is called class to basic type conversion. The constructor functions do not support this operation. This type of conversion takes place in casting operator. The casting operator function is also called as conversion function. The syntax of casting operator function is

```
operator typename()
{
        ……..
        //function body
        ……..
}
```

This function converts a class type data to typename. For example- the operator double() converts class object to type double. The operator int() converts a class type object to type int and so on.

The casting operator function should satisfy the following conditions

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

```
class Stock
{
        int items;
        float price;
        public:
        Stock(int a,float p)
        {
                items = a;
                price = p;
        }
        void putdata()
        {
                cout<<"Items: "<<items<<"\n";
                cout<<"Price: "<<price<<"\n";
        }
        operator float()
        {
                return (items*price);
        }

};
main()
{
        Stock s(45,2.5);
```

```
        float total_value;
        total_value = s;
        cout<<"\nData of s ";
        s.putdata();
        cout<<"Total float value = "<<total_value;
        getch();
        return 0;
}
```

Another example

```
class DistConv
{
        private:
        int kilometers;
        double meters;
        static double kilometersPerMile;
        public:
   // This function converts a built-in type (i.e. miles) to the
   // user-defined type (i.e. DistConv)
        DistConv(double mile)  // Constructor with one argument
        {
                double km = kilometersPerMile * mile ;    // converts miles to
                                //kilometers

                kilometers = int(km);           // converts float km to
                        //int and assigns to kilometer

                meters = (km - kilometers) * 1000 ;    // converts to meters
        }
        DistConv(int k, float m)   // constructor with two arguments
        {
                kilometers = k ;
                meters = m ;
        }
   // ********Conversion Function************
   operator double()              // converts user-defined type i.e.
                           // DistConv to a basic-type
   {                         // (double) i.e. meters
      double K = meters/1000 ;       // Converts the meters to
                              // kilometers
      K += double(kilometers) ;       // Adds the kilometers
      return K / kilometersPerMile ;    // Converts to miles
   }
```

```
    void display(void)
    {
       cout << kilometers << " kilometers and " << meters << " meters" ;
    }
}; // End of the Class Definition

double DistConv::kilometersPerMile = 1.609344;

int main(void)
{
    DistConv d1 = 5.0 ;              // Uses the constructor with one argument


    DistConv d2( 2, 25.5 );          // Uses the constructor with two arguments
    double ml = double(d2) ;       // This form uses the conversion function
                            // and converts DistConv to miles
    cout << "2.255 kilometers = " << ml << " miles\n" ;
    ml = d1 ;                       // This form also uses conversion function
                            // and converts DistConv to miles

    d1.display();
    cout << " = " << ml << " miles\n" ;
    getch();
}

/*Output
2.255 kilometers = 1.25859 miles
8 kilometers and 46.72 meters = 5 miles*/
```

From One Class to Another Class Type

When a data of one class type is converted into data of another class type, it is called conversion of one class to another class type. For example-

```
objx = objy;
```

here objx is an object of class X and objy is object of class Y. The class Y type data is converted to the class X type data and converted value is assigned to the objx. Since the conversion takes place from class Y to class X, Y is known as source class and X is known as destination class. This type of conversion is carried out by either constructor or a conversion function. Then how do we decide which form to use? It depends upon where we want the type conversion function to be located in the source class or in the destination class.

We know that the casting operator function

```
operator typename()
```

Converts the class object of which it is a member to typename. The typename may be a built-in type or user-defined type one. In the case of conversion between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. This implies that the argument belongs to the source class and is passed to the destination class for conversion. This makes it necessary that the conversion constructor be placed in the destination class.

```
class Kilometers
{
        private:
        double kilometers;
        public:
         Kilometers(double kms)
        {
                kilometers = kms;
        }
        void display()
        {
                cout << kilometers << " kilometeres";
        }
        double getValue()
        {
                return kilometers;
        }
};

class Miles
{
        private:
        double miles;
        public:
        Miles(double mls)
        {
                miles = mls;
        }
        void display()
        {
                cout << miles << " miles";
        }
        operator Kilometers()
```

```
        {
                return Kilometers(miles*1.609344);
        }
        Miles(Kilometers km)
        {
                miles = km.getValue()/1.609344;
        }
};


int main(void)
{
   /*
    * Converting using the conversion function
    */
   Miles m1 = 100;
   Kilometers k1 = m1;

   m1.display();
   cout << " = ";
   k1.display();
   cout << endl;

   /*
    * Converting using the constructor
    */
   Kilometers k2 = 100;
   Miles m2 = k2;    // same as:  Miles m2 = Miles(k2);
   k2.display();
   cout << " = ";
   m2.display();
   cout << endl;
   getch();
   return 0;
}
/*Output
100 miles = 160.934 kilometeres
100 kilometeres = 62.1371 miles
*/


class invent1
{
        int code;
        int items;
        float price;
```

```
        public:
        invent1(int a,int b,float c)
        {
                code = a;
                items = b;
                price = c;
        }
        void putdata()
        {
                cout<<"Code: "<<code<<"\n";
                cout<<"Items: "<<items<<"\n";
                cout<<"Value: "<<price<<"\n";
        }
        int getcode()
        {
                return code;
        }
        int getitems()
        {
                return items;
        }
        int getprice()
        {
                return price;
        }
        operator float()
        {
                return (items*price);
        }

};

class invent2
{
        int code;
        float value;
        public:
        invent2()
        {
                code = 0;
                value = 0;
        }
        invent2(int x,float y)
        {
                code = x;
                value = y;
```

```
        }
        void putdata()
        {
                cout<<"Code: "<<code<<"\n";
                cout<<"Value: "<<value<<"\n";
        }
        invent2(invent1 p)
        {
                code = p.getcode();
                value = p.getitems() * p.getprice();
        }
};

main()
{
        clrscr();
        invent1 s1(100,5,140.0);
        invent2 d1;
        float total_value;
        /*invent to float*/
        total_value = s1;
        /*invent1 to invent2*/
        d1 = s1;
        cout<<"Product details - invent1 type"<<"\n";
        s1.putdata();
        cout<<"Stock value"<<"\n";
        cout<<"value = "<<total_value<<"\n";
        cout<<"Product details-invent2 type"<<"\n";
        d1.putdata();
        getch();
        return 0;
}
```

Lab Sheet-2

1. Write a program to overload += operator. (You can overload this operator using Distance class as d1 += d2)

2. Write a program to overload = operator (It is already overloaded in C++). Use Distance class to test the program.

3. Write a program to overload ++ operator using friend function.

4. Create a class called **Length** that has data members meter and centimeter. Overload + operator to add two objects of class Length. (For example L3 = L1 + L2). Also facilitate the operations like L4 = L1 + 5 and L5 = 5 + L4 where L1, L2, L3, L4 and L5 are objects of class Length. Use constructors and member functions to initialize and display values.

5. Write a conversion routine in c++ that can convert user-defined data distance to basic data float. Assume that the class distance contains two data members (feet (integer type) and inch (floating point type). NOTE 1-meter = 3.33 feet and 1 feet = 12 inches)

6. Define a class to hold rectangular co-ordinates, i.e. x and y co-ordinates. Let P1 and P2 be the objects of this class where P1 is initialized to (20, 30). Facilitate the operation P2 = P1++ in such a way that the value in P2 is (21, 31) afterward.

7. Write a program to overload + operator to concatenate two strings.

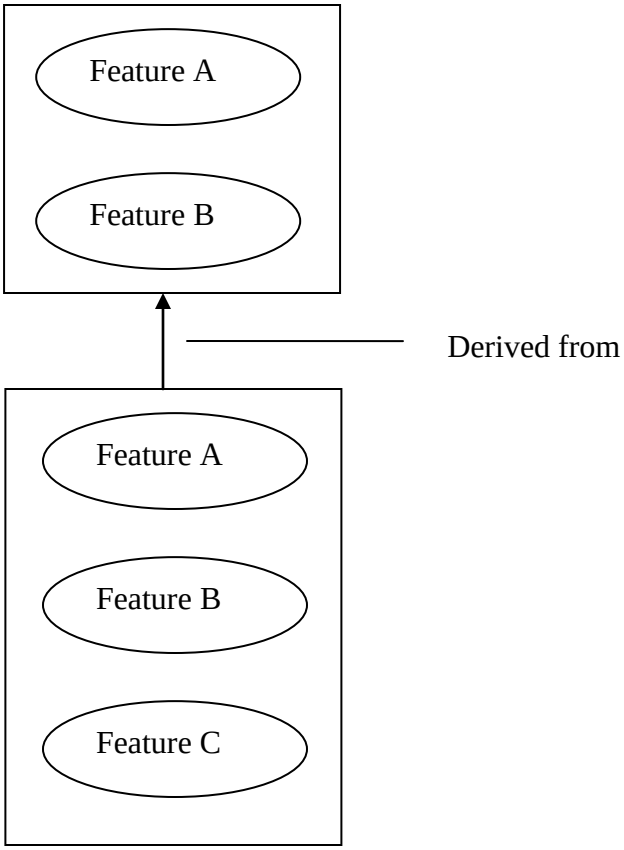
Inheritance

Inheritance is the most powerful feature of object-oriented programming. Inheritance is the process of creating new classes, called derived classes, from existing classes or base classes. The class inherits all the capabilities of the base class but can add refinements of its own.

When new class is created based on some other class using inheritance, the newly created class is called the derived class or sub-class, while the class on which it is based is called base class or superclass.

Inheritance is also called a 'kind of relationship'. Inheritance supports the concept of '**reusability**'. Once a class has been written and tested, its features can be adapted by other programmers whenever required.

Base class



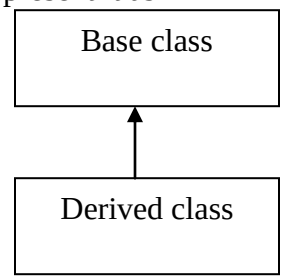
Derived class

Fig. Inheritance

Types

There are 5 types of inheritance

1) Single Inheritance – when a class is derived from only one base class, then it is called single inheritance. We can represent it as

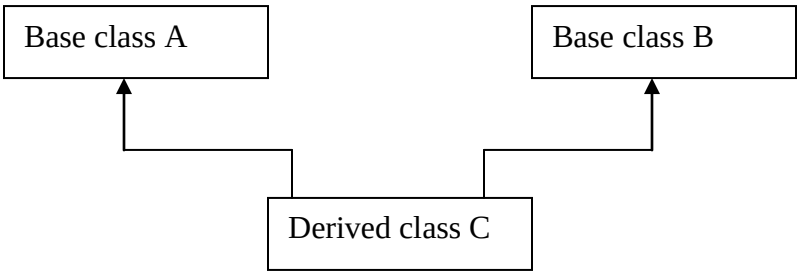


Eg-

```
class A       // base class
{
        private:
```

```
        {
                ….  // body part
                …
        }
        public:
        {
                …..// body part
                …..
        }
};
class B : public A    // derived class B
{
        ……
        …..
};
```

2) Multiple Inheritance: When a class is derived from two or more base classes, it is called multiple inheritance. It can be represented as,

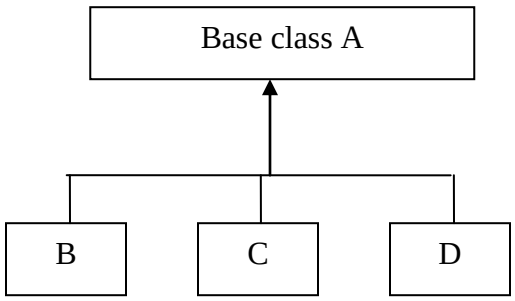


Eg-

```
class A
{
  …….
   ……
};

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

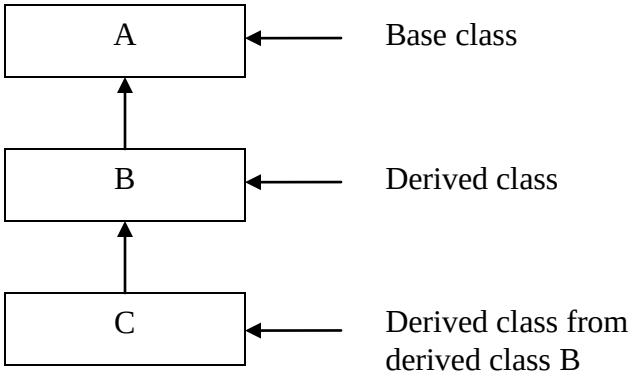
```
        .......
        ......
    };
```
Here two base classes A and B have been created and a derived class C is created from both base classes.

3) Hierarchical Inheritance: When two or more than two classes are derived from one base class, it is called hierarchical inheritance. It can be represented as



Eg-
```
Class A
{
  ..........
  ..........
};
class B : public A
{
   ........
    ........
};
class C : public A
{
  ........
 ........
};
```
Here two classes B and C are derived from same base class A.

4) Multilevel Inheritance: The mechanism for deriving a class from another derived class is known as multilevel inheritance and can be presented as

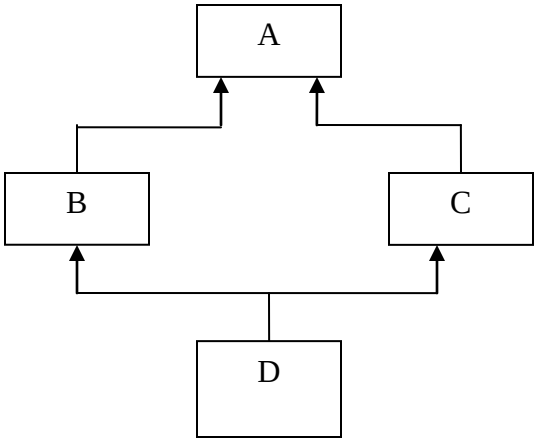


E.g.-

```
Class A
{
  …..
  …..
};
classs B : public A
{
  …….
  ……
};
class C : public B
{
  …..
  …..
};
```

In this example class B is derived from base class A and class C is derived from derived class B.

5) Hybrid Inheritance: This inheritance is the combination of multiple and hierarchical inheritance, it can be represented as



E.g-

```
class student
{
      ……
      …..
};
class test : public student
{
    …….
    …….
};
class sport : public student
{
    ……
    …...
};
class result : public test, public sport
{
   ……
   ….
};
```

In the example, there is one base class student. The class test and sport are derived from student and class result is derived from both test and sport classes. This shows the combination of multiple and hierarchical inheritance.

Sub-class Definition

A subclass can be defined by specifying its relationship with base class along with its own details. General form of defining sub class is

```
class    subclass-name : visibility-mode     baseclass-name
{
  ……….
  ……
};
```

The colon indicates that the inheritance has been used. i.e. the sub class has been derived from base class. The visibility mode specifies whether the features of the base class are derived privately, publicly or protectedly. This is optional and if absent, the default is private.
The three visibility modes are

- public
- private
- protected

When the base class is ***publicly inherited***, all the public members of the base class become public members of the derived class. So, they can be accessed through the objects of the derived class. All the protected members become protected members of the derived class.

When the base class is ***privately inherited***, all the public and protected members of the base class become private members of the derived class. So, these can not be accessed outside the class directly through the derived class object, but might be accessed through public functions in the derived class. Like general private members, these members can be used freely within the derived class.

NOTE: private members can not be inherited at all. Only the public and protected ones can be inherited.

When a member is defined with ***protected*** access specifier, these members can be accessed from that class and also from the derived class of this base class. But can not be accessed from any other function or class. i.e. protected members act as public for derived class and private for other classes.

When base class is derived using ***protected*** mode, all the protected and public members of the base class become protected members of the derived class. This means, like a private inheritance, these members can not be directly accessed through object of the derived class. But can be used freely within the derived class. Whereas, unlike a private inheritance, they can still be inherited and accessed by subsequent derived classes. In other words, protected inheritance does not end a hierarchy of classes, as private inheritance does.

Overriding base class members
In C++, a base class member can be overridden by defining a derived class member with the same name as that of the base class member. Consider the program

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

```
class aclass
{
      public:
         void disp(void)
          {
              cout<<"Base"<<endl;
          }
};
class bclass : public aclass
{
      public:
         void disp(void)
          {
               cout<<"Derived"<<endl;
           }
};
void main()
{
      bclass Bvar;
      Bvar.disp();
}
```

Here, the function disp() is overridden.

- If the function is invoked from an object of the derived class, then the function in the derived is executed.
- If the function is invoked from an object of the base class, then the base class member function is invoked.

Ambiguities in Multiple Inheritance

When a class inherits from multiple base classes, a whole part of ambiguities creep in. for eg- what happens when two base classes contain a function of the same name? Consider the following program,

```
#include<iostream.h>
class base1
{
    public:
        void disp(void)
         {
             cout<<"Base1"<<endl;
          }
};
class base2
{
    public:
```

```
        void disp(void)
         {
               cout<<"Base2"<<endl;
          }
};
class derived : public base1, public base2
{
   // Empty
};
void main()
{
   derived Dvar;
    Dvar.disp();   // Ambiguous
}
```

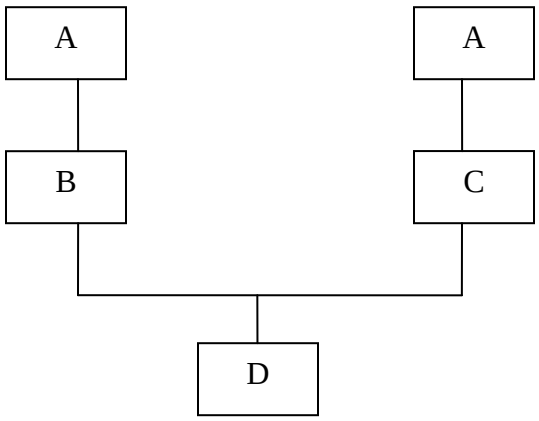
Here, the reference to disp() is ambiguous because the compiler does not know whether disp() refers to the member in class base1 or base2. This ambiguity can be resolved using the scope resolution operator as

```
void main()
{
     derived Dvar;
     Dvar.base1::disp();
     Dvar.base2::disp();
}
```

This ambiguity can also be resolved by overriding. That is, the members can be redefined in the derived class. For eg-

```
class base1
{
  ……
  …...
};
class derived : public base1, public base2
{
     public:
         void disp(void)
         {
             base1::disp();
             base2::disp();
             cout<<"Derived class"<<endl;
           }
};
void main()
{
        derived Dvar;
         Dvar.disp();     // Not ambiguous
```

}
Another ambiguity that arises in multiple inheritance is the possibility of the derived class having multiple copies of the same base class. Consider the following diagram



Inheritance from a base class via different paths

In the above figure, class D inherits from two base classes namely B and C. B and C in turn are derived from class A. As a result, class D would have two copies of class A. There are two ways of resolving this ambiguity. One way is to the scope resolution operator and the other way is to use virtual base classes.

Containership
When a class contains object of another class as its member data, it is termed as containership. The class which contains the object is called container class. Containership is also termed as "class within class".

```
class A
{
   ……
   ……
};
class B
{
  …..
  A obj1;
  …..
};
```

Here, class B contains object of class A. So B is the container class.

'containership' is also called **'has-a'** relationship.

In some situations, inheritance and containership relationship serves similar purpose. Containership is useful with classes that act like a data type. The object of these classes can be used almost like other variables in the class.

In inheritance, if a class **B** is derived from a class **A**, then "**B** is a kind of **A**". This is because B has all the characteristics of A, and in addition some of its own. So, inheritance is often called a **"kind of"** relationship.

```
class Manager
{
  char name[20];
  int age;
  public:
  void getdata()
  {
      cin>>name>>age;
  }
  void putdata()
  {
      cout<<name<<age;
  }
};
class Employee    // Employee is container class
{
  char department[20];
  Manager m;    // Object of class Manager
  public:
  void getdata()
  {
    m.getdata();
    cin>>department;
  }
  void putdata()
  {
    m.putdata();
    cout<<department;
  }
};
void main()
{
   Employee e;
   e.getdata();
   e.putdata();
   getch();
}
```

Abstract Class

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class. It is a design concept in program development and provides a base upon which other classes may be built.

Virtual Base Class

The principle behind the virtual base class is very simple. When the same class is inherited more than once via multiple paths, multiple copies of the base class members are created in memory. By declaring the base class inheritance as virtual, only one copy of the base class is inherited. A base class inheritance can be specified as a virtual using the virtual qualifier.

```
class A
{
  …..
  …..
};
class B1 : virtual public A
{
  ….
  ….
};
class B2 : public virtual A
{
  ……
  …...
};
class C : public B1, public B2
{
  ……
  ……
};
```

Keywords 'virtual' and 'public' can be used in either way.

When a class is made virtual base class, only one copy of that is inherited, regardless of how many number of inheritance path exists between the virtual base class and derived class. Since there is only one copy, there is no ambiguity.

Constructors in Derived Class

One important thing to note is that as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. While applying inheritance, we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When

both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them onto the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor. Example-

```
#include<iostream.h>
class alpha
{
  int x;
  public:
  alpha(int p)
  {
    x = p;
    cout<<"\nAlpha initialized";
  }
  void show_x()
  {
    cout<<"\nX = "<<x;
  }
};
class beta
{
  float y;
  public:
  beta(float q)
  {
      y = q;
      cout<<"\nBeta initialized";
  }
  void show_y()
  {
     cout<<"\nY = "<<y;
  }
};
class gamma : public alpha, public beta
{
  int m,n;
  public:
```

```
  gamma(int a,float b,int c,int d): beta(b),alpha(a)
  {
    m = c;
    n = d;
    cout<<"\nGamma initialized";
  }
  void show_mn()
 {
     cout<<"\nM = "<<m<<"\nN = "<<n;
 }
};
void main()
{
  gamma g(10,5.5,50,60);
  g.show_x();
  g.show_y();
  g.show_mn();

}
```

Constructor and destructors in derived class: order of executions

Situation remains understandable until both the base and its derived class have Constructors and/or Destructors. Since the derived class contains more than one Constructors and/or Destructors, it becomes confusing which one will be called when.

This is because when an object the inherited class is constructed both the constructors (base's and its own) should be invoked and same applies when it gets destructed.

This article will clear all this!

Consider the following example program:

```
// -- INHERITANCE --
// Constructors, Destructors
// and Inheritance
#include<iostream.h>

// base class
class base
{
public:
  base(){cout<<"Constructing Base\n";}
  ~base(){cout<<"Destructing Base\n";}
```

```
};

// derived class
class derived:public base
{
public:
  derived(){cout<<"Constructing Derived\n";}
  ~derived(){cout<<"Destructing Derived\n";}
};

void main(void)
{
  derived obj;

  // do nothing else, only
  // construct and destruct
  // the inherited class object
}
```

OUTPUT:

```
Constructing Base
Constructing Derived
Destructing Derived
Destructing Base
Press any key to continue
```

So here is the general rule:

Constructors are called in the order of derivation and Destructors in the reverse order.

One more example will clear the confusions, if any.:

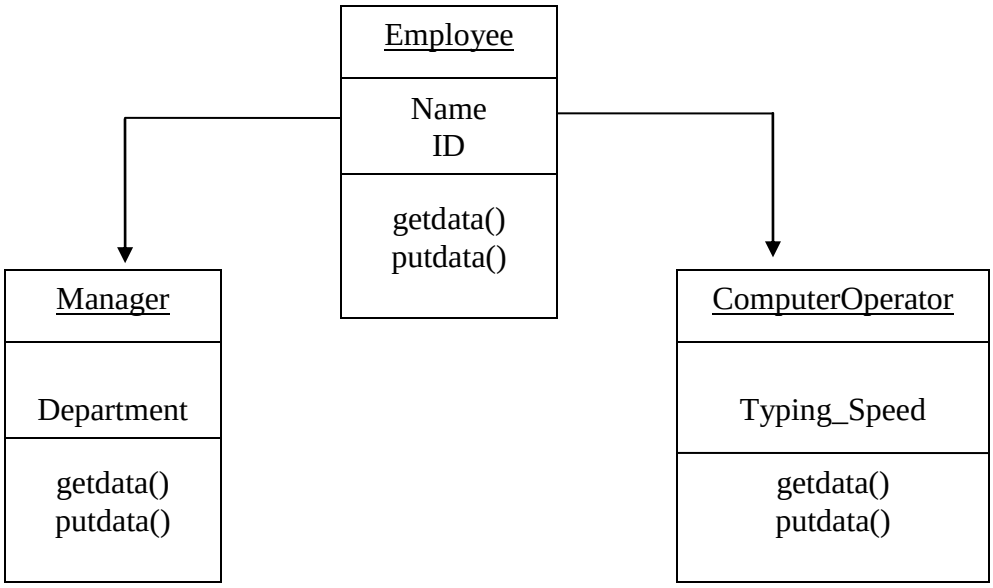
```
// -- INHERITANCE --
// Constructors, Destructors
// and Inheritance
#include<iostream.h>

// base class (1)
class base
{
public:
  base(){cout<<"Constructing Base\n";}
  ~base(){cout<<"Destructing Base\n";}
};
```

```
// derived class
// derived form 'base'
class derived1:public base
{
public:
  derived1(){cout<<"Constructing Derived1\n";}
  ~derived1(){cout<<"Destructing Derived1\n";}
};

// derived from a derived class
// 'derived1'
class derived2:public derived1
{
public:
  derived2(){cout<<"Constructing Derived2\n";}
  ~derived2(){cout<<"Destructing Derived2\n";}
};

void main(void)
{
  derived2 obj;

  // do nothing else, only
  // construct and destruct
  // the inherited class object
}
```

OUTPUT:

```
Constructing Base
Constructing Derived1
Constructing Derived2
Destructing Derived2
Destructing Derived1
Destructing Base
```

Lab Sheet - 3



1. Write a C++ program to represent the above inheritance scheme. Also write a main() function to test the classes, Manager and ComputerOperator, by creating their objects, taking input and displaying the corresponding values.

2. Imagine a college hires some lecturers. Some lecturers are paid in period basis, while others are paid in month basis. Create a class called **lecturer** that stores the **ID**, and the **name of lecturers**. From this class derive two classes: **PartTime**, which adds payperhr (type float); and **FullTime**, which adds paypermonth (type float). Each of these three classes should have a **readdata()** function to get its data from the user, and a **printdata()** function to display its data.

   Write a **main()** program to test the **FullTime** and **PartTime** classes by creating instances of them, asking the user to fill in their data with **readdata()**, and then displaying the data with **printdata()**.

3. An industry seals lorry and taxi. Create a class **Automobile** that stores production date and price. From this class derive another two classes: **Lorry**, which adds weight capacity in kilogram and **Taxi**, which adds seat-capacity in number. Each of these classes should have member functions to get data and set data. Use user-defined constructors to initialize these objects.

4. Create a class called **cricketer** with member variables to represent name, age and no of matches played. From this class derive two classes: **Bowler** and **Batsman**. Bowler class has no_of_wickets as member variable and Batsman class has no_of_runs and centuries as member variables. Use appropriate member functions in all classes to read and display respective data.

5. Define a base class **Shape** having data member radius (int). Derive new classes called **Circle** and **Sphere** from this class. Write methods to compute the area of circle and sphere.

6. Create classes **Book** having data members name of author (string), price (float) and class **Stock** having data members number of books (int) and category (string). Create another class **Library** which derives from both the classes Book and Stock. All the classes should have functions having same name. Write program to test these classes.

Virtual Function and Run Time Polymorphism

Pointers

Pointers have a reputation for being hard to understand. One important use for pointers is in the dynamic allocation of memory, carried out in C++ with the keyword new and its partner delete

Addresses (Pointer Constants)

Every byte in the computer's memory has an address. Addresses are numbers, just as they are for houses on a street.

The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575; for 16 MB of memory, it is 16,777,215.

The Address of Operator &

You can find out the address occupied by a variable by using the address of operator &.

New and Delete Operator

Pointer provides the necessary support for C++ powerful dynamic memory allocation system. Dynamic allocation is the means by which a program can obtain memory while it is running.

For eg- int arr[100];

reserves memory for 100 integers. Arrays are a useful approach to data storage, but they have a serious drawback. We must know at the time we write the program how big the array will be and it is not always possible to predict what the size of the array will be. It would be desirable to start the program and then allocates memory as the need arises. This capability is provided by the new operator. This versatile operator obtains memory from the operating system and returns a pointer to the starting point. The syntax for the new operator is
<variable> = new <type>;
where <variable> = pointer variable
<type> = char, int, float and so on

type of variable mentioned on the left hand side and the type mentioned on the right hand side should match. For eg-
char * cptr;
cptr = new char;
int *iptr;
iptr = new int;

the syntax of new operator can also be modified to allocate memory of varying requirements. For eg-
char *cptr;
cptr = new char[10];
allocates 10 bytes of memory and assigns the starting address to cptr.

Delete Operator
If our programs reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system.
Deleting the memory does not delete the pointer that points to it and does not change the address value in the pointer. However, this address is no longer valid, the memory it points to may be changed to something entirely different but we do not use pointers memory that has been deleted.
Syntax:
delete <variable>;
Where <variable> = pointer variable
Eg- student *ps;
ps = new student;
delete ps;
If we are deleting an array, we use bracket following delete. For eg-
delete[] cptr;

//Example program that makes use of new and delete operator
#include<iostream.h>
#include<conio.h>
class test

```
{
  int x;
  public:
  void input()
  {
        cin>>x;
  }
  void display()
  {
        cout<<x;
  }
};
main()
{
  test *t = new test;
  t->input();
  t->display();
  delete t;
  getch();
  return 0;
}
```

Polymorphism

In the programming sphere, polymorphism is broadly divided into two parts- the first part being static polymorphism- exhibited by overloaded functions and the second being dynamic polymorphism exhibited by late binding.

Static Polymorphism

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions on the basis of number, type, and sequence of their arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to the calls at compile time. This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions. The resolution is on the basis of number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration.

```
void add(int, int);
void add(float, float);
```

Now, if the function add() is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the

program is executed. The term late binding refers to the resolution of the function to their associated methods at run time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context. The compiler is unable to bind a call to a method since resolution depends on the context of the call.

Static binding is considered to be more efficient and dynamic binding more flexible.

Virtual Functions

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. A function is made virtual by placing the keyword **virtual** before its normal declaration.

Normal Member Functions Accessed with Pointers

What will happen when a base class and the derived classes all have functions with the same name, and we access these functions using pointers but without using virtual functions? Here is the program for this.

```
#include<iostream.h>
#include<conio.h>
class Base
{
        public:
        void show()
        {
                cout<<"Base\n";
        }
};
class Derv1 : public Base
{
        public:
        void show()
        {
                cout<<"Derv1\n";
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
                cout<<"Derv2\n";
        }
};
int main()
```

```
{
        Derv1 d1;
        Derv2 d2;
        Base *ptr; // pointer to base class
        ptr = &d1;
        ptr->show();
        ptr = &d2;
        ptr->show();
        getch();
        return 0;
}
```

Output of the program
Base
Base
The function in the base class is always executed. The compiler ignores the contents of the pointer ptr and chooses the member function that matches the type of the pointer.

Virtual Member Functions Accessed with Pointers
Let's make a single change in the above program. We will place the keyword **virtual** in front of the declarator for the **show()** function in the base class.

```
#include<iostream.h>
#include<conio.h>
class Base
{
        public:
        virtual void show()
        {
                cout<<"Base\n";
        }
};
class Derv1 : public Base
{
        public:
        void show()
        {
                cout<<"Derv1\n";
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
                cout<<"Derv2\n";
        }
```

```
};
int main()
{
        Derv1 d1;
        Derv2 d2;
        Base *ptr;
        ptr = &d1;
        ptr->show();
        ptr = &d2;
        ptr->show();
        getch();
        return 0;
}
```

Output of the program

Derv1

Derv2

Now, as you can see, the member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of Derv1 to that of Derv2, and the particular instance of show() that is executed also changes. So the same function call,

```
ptr->show();
```

executes different functions, depending on the contents of ptr. The compiler selects the function according to the contents of the pointer ptr, not on the type of the pointer. Here, the compiler does not know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class. Which version of show() does the compiler call? In fact the compiler does not know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointing to by ptr, the appropriate version of show() will be called, exhibiting late binding.

Abstract Classes and Pure Virtual Functions

An abstract class is one that is not used to create objects. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. A class is made an abstract by placing at least one pure virtual function in the class. A pure virtual function is one with expression = 0 added to the declaration. i.e. a pure virtual function can be declared by equating it to zero.

For example, to make show() function virtual we write

```
virtual void show ()= 0;  // pure virtual function
```

the equal sign here has nothing to do with assignments, the value 0 is not assigned to anything. The =0 syntax is simply how we will tell the compiler that a function will be pure.

Once we have placed a pure virtual function in the base class, then we must override it in all the derived classes from which we want to instantiate objects. If a class does not override the pure virtual function, then it becomes an abstract class itself, and we can not instantiate objects from it. For consistency, we make all the virtual functions in the base class pure.
We can not create objects of the abstract class. However, we can create pointers to an abstract class. This allows an abstract class to be used as a base class, pointers to which can be used to select the proper virtual function.

```
//Program: Pure virtual function
#include<iostream.h>
#include<conio.h>
class Base
{
        public:
        virtual void show() = 0;
};
class Derv1 : public Base
{
        public:
        void show()
        {
                cout<<"Derv1\n";
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
                cout<<"Derv2\n";
        }
};
int main()
{
        //Base b; // can't make object of abstract class
        Base *arr[2];
        Derv1 d1;
        Derv2 d2;
        arr[0] = &d1;
        arr[1] = &d2;
        arr[0]->show();
        arr[1]->show();
        getch();
        return 0;
}
```

Output of the program
Derv1
Derv2

```
//Example program: pure virtual function
#include<iostream.h>
#include<conio.h>
class person
{
        protected:
        char name[20];
        public:
        void getName()
        {
                cin>>name;
        }
        void putName()
        {
                cout<<endl<<name;
        }
        virtual void getData() = 0;
        virtual int isOutstanding() = 0;
};
class student : public person
{
        private:
        float gpa;
        public:
        void getData()
        {
                person::getName();
                cout<<"\nEnter students gpa";
                cin>>gpa;
        }
        int isOutstanding()
        {
                return (gpa > 3.5) ? 1 : 0;
        }
};
class professor : public person
{
        private:
        int numpubs;
        public:
        void getData()
        {
```

```
                person :: getName();
                cout<<"\nEnter number of publications";
                cin>>numpubs;
        }
        int isOutstanding()
        {
                return (numpubs > 100)?1 : 0;
        }
};
main()
{
        person *persptr[100];
        int n = 0;
        char choice;
        do
        {
                cout<<"\nEnter student or professor(s/p)";
                cin>>choice;
                if(choice == 's')
                        persptr[n] = new student;
                else
                        persptr[n] = new professor;
                persptr[n++]->getData();
                cout<<"\nAnother record? ";
                cin>>choice;
        }while(choice == 'y');

        for(int j = 0;j < n; j++)
        {
                persptr[j]->putName();
                if(persptr[j]->isOutstanding())
                cout<<"\nThis person is outstanding";
        }
        getch();
        return 0;
}
```

Friend class

The member functions of a class can all be made friends at the same time when we make the entire class a friend. Consider the following program.

```
#include<iostream.h>
#include<conio.h>
class alpha
{
        int data;
        public:
```

```
        alpha()
        {
                data = 10;
        }
        friend class beta;  // beta is friend class
};
class beta
{
        public:
        void func1(alpha a)
        {
                cout<<a.data;
        }
        void func2(alpha a)
        {
                cout<<a.data;
        }

};
main()
{
        alpha a;
        beta b;
        b.func1(a);
        b.func2(a);
        getch();
        return 0;
}
```

In class **alpha** the entire class **beta** is defined as a friend. Now all the member functions of **beta** can access the private data of **alpha**.

The this Pointer

The member functions of every object have access to a sort of magic pointer named **this**, which points to the object itself. Consider the program

```
#include<iostream.h>
#include<conio.h>
class MyClass
{
        public:
        void TestThisPointer()
        {
                cout<<"\nMy object's address is"<<this;
        }
};
```

```
main()
{
        MyClass m1,m2;
        m1.TestThisPointer();
        m2.TestThisPointer();
        getch();
        return 0;
}
```

Accessing Member Data with this

When we call a member function, it comes into existence with the value of **this** set to the address of the object for which it was called. The **this** pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to.

```
class MyClass
{
        int x;
        public:
        void test()
        {
                this->x = 10;
                cout<<this->x;
        }
};
main()
{
        MyClass m;
        m.test();
        getch();
        return 0;
}
```

This program simply prints the value 10. The test() member function accesses the variable **x** as

```
this->x
```

This is exactly the same as referring to **x** directly.

Using this for Returning Values

A more practical use for **this** is in returning values from member functions and overloaded operator.

```
class alpha
{
        int data;
```

```
        public:
        alpha()
        {}
        alpha(int x)
        {
                data = x;
        }
        void display()
        {
                cout<<data<<endl;
        }
        alpha& operator = (alpha &a)
        {
                data = a.data;
                return *this;
        }
};
main()
{
        alpha a1(50);
        alpha a2;
        a2 = a1; // calls overloaded operator =
        a1.display();
        a2.display();
        getch();
        return 0;
}
```

//using **this** pointer for returning value from member function

```
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
```

```
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance compare(Distance);
};
Distance Distance :: compare(Distance d2)
{
        float dist1 = meter + (float)centimeter/100;
        float dist2 = d2.meter + (float)d2.centimeter/100;
        if(dist1 < dist2)
                return *this;
        else
                return d2;
}
main()
{
        Distance d1(4,50);
        Distance d2,d3,d4;
        d2.getDist();
        d4 = d1.compare(d2);
        cout<<"Small lenght is: ";
        d4.show();
        getch();
        return 0;
}
```

File I/O Operations in C++

Working with Files and File I/O operations

A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files. A program typically involves either or both of the following kinds of data communication:

1. Data transfer between the console unit and the program. (already discussed)
2. Data transfer between the program and a disk file.

The I/O system of C++ handles file operations which are very much similar to the console I/O operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as **input stream** and the one that receives data from the program is known as **output stream**. In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file.

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

The I/O system of C++ contains a set of classes that define the file handling methods. These include **ifstream**, **ofstream** and **fstream**. These classes are derived from fstreambase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in **fstream** and therefore we must include this file in any program that uses files.

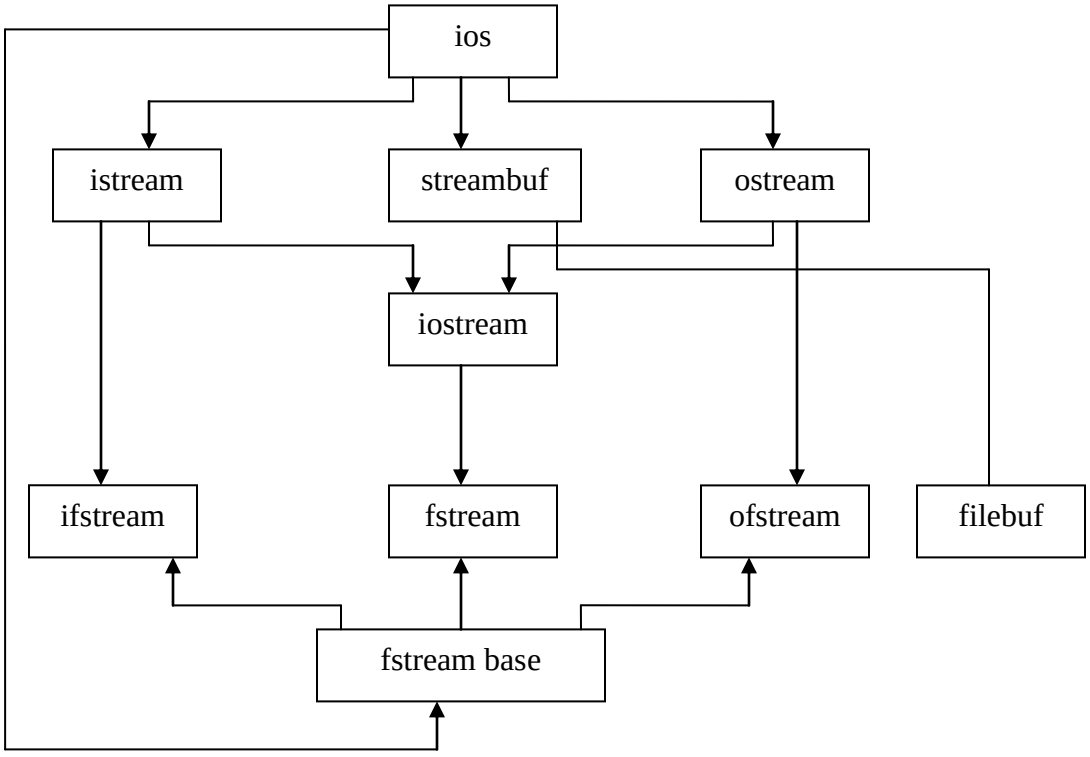


Fig. stream classes for file I/O operations

ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.

Opening and Closing a File
We must first create a file stream and then link it to the file name. A file stream can be defined using the classes **ifstream**, **ofstream**, and **fstream**. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways

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

Opening Files Using Constructor
Opening a file using constructor involves the following steps:

1. Create a file stream object to mange the stream using the appropriate class. That is to say, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
2. Initialize the file object with the desired filename.

For example: ofstream file("student"); //write only
ifstream file("employee"); // read only

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
        char name[20];
        int age;
        char sec;
        ofstream ofile("Student.txt");
        cout<<"\nEnter name, age section: ";
        cin>>name>>age>>sec;
        ofile<<name<<endl;
        ofile<<age<<endl;
        ofile<<sec;
        ofile.close();
        ifstream ifile("student.txt");
        ifile>>name;
        ifile>>age;
        ifile>>sec;
        cout<<name<<"\t"<<age<<"\t"<<sec;
        ifile.close();
        getch();
}
```

When a file is opened for writing only, a new file is created if there is no file of that name. If a file by that name exists already, then its contents are deleted and the file is presented as a new one.

Opening Files Using open()

The function open() can be used to open multiple files that use the same stream object. In such cases, we may create a single stream object and use it to open each file in turn. This is done as

```
file-stream-class    stream-object;
stream-object.open("filename");
```

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

```
#include<conio.h>
main()
{
        char name[20];
        int age,salary;
        char sec;
        ofstream ofile;
        ofile.open("student.txt");
        cout<<"\nEnter name, age section: ";
        cin>>name>>age>>sec;
        ofile<<name<<endl;
        ofile<<age<<endl;
        ofile<<sec;
        ofile.close();
        ofile.open("teacher.txt");
        cout<<"\nEnter name and salary of teacher: ";
        cin>>name>>salary;
        ofile<<name<<endl;
        ofile<<salary;
        ofile.close();
        getch();
}
```

File Modes with open()

The open function can take two arguments, the second one for specifying the file mode. The general form of the function open with two arguments is

stream-object.open(“filename”, mode);

The second argument (called file mode parameter) specifies the purpose for which the file is opened.

Parameter	**Meaning**
ios :: app	append to end of file
ios :: ate	go to end of file on opening
ios :: in	open file for reading only
ios :: out	open file for writing only
ios :: binary	binary file
ios :: nocreate	open fails if the file does not exists
ios :: noreplace	open fails if the file already exists
ios :: trunc	delete the contents of the file if it exists

The mode can combine two or more parameters using the bitwise OR operator (|).

put() and get() Functions

The function put() writes a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        char ch;
        fstream file;
        file.open("myfile.txt",ios::out); // write only
        cin.get(ch);
        while(ch != '\n')
        {
                file.put(ch);
                cin.get(ch);
        }
        file.close();
        file.open("myfile.txt",ios::in);  // read only
        while(file)
        {
                file.get(ch);
                cout<<ch;
        }
        file.close();
        getch();
}
```

write() and read() Functions

The functions write() and read(), unlike the functions put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. For example- an **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit **int** will take four bytes to store it in the character form. **write()** and **read()** functions take two arguments. The first is the address of the variable, and the second is the length of that variable in bytes. The address of the variable must be cast to type char* (pointer to character type).

```
//writing array of integers using write()
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        int x[] = {100,200,300,400};
        int i;
        fstream file;
```

```
	file.open("myfile.dat",ios::out|ios::binary);
	file.write((char*)&x,sizeof(x));
	file.close();
	for(i = 0;i < 4;i++)
	x[i] = 0;
	file.open("myfile.dat",ios::in|ios::binary);
	file.read((char*)&x,sizeof(x));
	for(i = 0;i < 4;i++)
	cout<<x[i];
	getch();
}
```

Writing and Reading Class Objects (Object I/O)

When writing an object we generally want to use binary mode. This writes the same bit configuration to disk that was stored in memory. The binary input output functions **write()** and **read()** are used for this.

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class person
{
	protected:
	char name[20];
	int age;
	public:
	void getData()
	{
		cin>>name>>age;
	}
	void showData()
	{
		cout<<name<<"\t"<<age<<endl;
	}
};

main()
{
	person p,q;
	fstream file;
	file.open("person.dat",ios::out|ios::binary);
	cout<<"\nEnter person's data: ";
	p.getData();
	file.write((char *)(&p),sizeof(p));
```

```
        file.close();
        file.open("person.dat",ios::in|ios::binary);
        file.read((char *)(&q),sizeof(q));
        q.showData();
        getch();
        return 0;
}
```

Writing Multiple Objects

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class person
{
        protected:
        char name[20];
        int age;
        public:
        void getData()
        {
                cin>>name>>age;
        }
        void showData()
        {
                cout<<name<<"\t"<<age<<endl;
        }
};
main()
{
        char ch;
        person p;
        fstream file;
        file.open("person.dat",ios::out|ios::in|ios::binary);
        do
        {
                cout<<"\nEnter person's data: ";
                p.getData();
                file.write((char *)(&p),sizeof(p));
                cout<<"Enter another person? ";
                cin>>ch;
        }while(ch == 'y');
        file.seekg(0);
        file.read((char *)(&p),sizeof(p));
        while(!file.eof())
        {
```

```
            p.showData();
            file.read((char *)(&p),sizeof(p));
        }
        cout<<endl;
        getch();
        return 0;
}
```

File Pointers

Each file object has associated with it two integer values called the *get pointer* and the *put pointer*. These are also called the current get position and the current put position, or simply the current position. These values specify the byte number in the file where writing or reading will take place. The **seekg()** and **tellg()** functions allow us to set and examine the get pointer, and the **seekp()** and **tellp()** functions perform these same actions on the put pointer.

Specifying the offset
We can move the file pointer to a desired location using the **seek** function. The argument to these functions represents the absolute position in the file. Syntax for this is:

```
seekg(offset, refposition);
seekp(offset, refposition);
```

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition takes one of the following three constants defined in the class **ios**

ios :: beg	start of file
ios :: cur	current position of the pointer
ios :: end	end of the file.

The **seekg()** function moves the associated file's *get pointer* while the **seekp()** function moves the associated file's *put pointer*. If **file** is an object of ofstream, then some some sample pointer offset calls and their actions are:

Seek call	**Action**
file.seekg(0,ios::beg);	go to start
file.seekg(0,ios::cur);	stay at the current position
file.seekg(0,ios::end);	go to the end of file.
file.seekg(m,ios::beg);	move to (m+1)th byte in the file
file.seekg(m,ios::cur);	go forward by m byte from the current position

```
// program using file pointers
#include<fstream.h>
```

```
#include<iostream.h>
#include<conio.h>
class person
{
	protected:
	char name[20];
	int age;
	public:
	void getData()
	{
		cin>>name>>age;
	}
	void showData()
	{
		cout<<name<<"\t"<<age<<endl;
	}
};
main()
{
	person p;
	fstream file;
	file.open("group.dat",ios::in|ios::binary);
	file.seekg(0,ios::end);
	int endposition = file.tellg();
	int n = endposition/sizeof(p);
	cout<<"\nNumber of records in file = "<<n;
	cout<<"\nEnter person number: ";
	cin>>n;
	int pos = (n-1)*sizeof(p);
	file.seekg(pos);
	file.read((char*)&p,sizeof(p));
	p.showData();
	cout<<"\nRecords in file are:\n";
	file.seekg(0,ios::beg);
	file.read((char*)&p,sizeof(p));
	while(!file.eof())
	{
		p.showData();
		file.read((char*)&p,sizeof(p));
	}
	file.close();
	getch();
	return 0;
}
```

```
// File copy program. This program copies the content of one file into another.
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#include<ctype.h>
void main()
{
        char ch;
        fstream file1,file2;
        file1.open("test1.txt",ios::out|ios::in);
        file1<<"welcome to object oriented programming";
        file1.close();
        file1.open("test1.txt",ios::in);
        file2.open("test2.txt",ios::out);
        while(file1)
        {
                file1.get(ch);
                ch = toupper(ch);
                file2.put(ch);
        }
        file1.close();
        file2.close();
        file2.open("test2.txt",ios::in);
        while(file2)
        {
                file2.get(ch);
                cout<<ch;
        }
        file2.close();
        getch();
}
```

Error Handling During File Operations

The class **ios** supports several member functions that can be used for handling error occurred during file operations. Some functions are

eof()	returns true if end of file is encountered while reading, otherwise return false.
fail()	returns true when an input or output operation has failed.
bad()	returns true if an invalid operation is attempted or any unrecoverable error has occurred.
good()	returns true if no error has occurred.

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

```
#include<fstream.h>
void main()
{
	char ch;
	fstream file;
	file.open("myfile.txt",ios::in);  // read only
	if(file.fail())
	{
		cout<<"\nCould not open the file";
	}
	else
	{
		while(file)
		{
			file.get(ch);
			cout<<ch;
		}
		file.close();
	}
	getch();
}
```

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
	char ch;
	fstream file;
	file.open("myfile.txt",ios::in);  // read only
	if(!file.good())
	{
		cout<<"\nCould not open the file";
	}
	else
	{
		while(file)
		{
			file.get(ch);
			cout<<ch;
		}
		file.close();
	}
```

```
        getch();
}
```

Namespace

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name. The format of namespaces is:

```
namespace identifier
{
entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

Using scope resolution operator:
In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the ”myNamespace” namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

Example program1

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

namespace first
{
  int var = 5;
}

namespace second
{
```

```
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.

Through keyword *using:*

The keyword using is used to introduce a name from a namespace into the current declarative region.

Example program2

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first
  cout << x << endl;
  cout << y << endl;
using namespace second
cout<<x<<endl;
cout<<y<<endl;
  return 0;
}
```

Templates

(Meaning to word: A document or file or entity having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used)

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.

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

A template is a way to specify generic code, with a placeholder for the type. Note that the type is the only "parameter" of a template, but a very powerful one, since anything from a function to a class (or a routine) can be specified in "general" terms without concerning yourself about the specific type.

Templates offer several advantages:

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.

- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Templates and Macros

In many ways, templates work like preprocessor macros, replacing the templated variable with the given type. However, there are many differences between a macro like this:

#define min(i, j) (((i) < (j)) ? (i) : (j))

and a template:

template<class T> T min (T i, T j) { return ((i < j) ? i : j) }

Here are some problems with the macro:

- There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.
- The i and j parameters are evaluated twice. For example, if either parameter has a post incremented variable, the increment is performed two times.
- Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging.

Templates in c++ comes in two variations
a) function templates
b) class templates

Class template

The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.
The general form of a class template is
Template <class T>
Class class_name
{
//class member with type T whenever appropriate
};
Example

```
#include <iostream.h>

template<class T>
class vec {
```

```
        T x;
        T y;

public:
        vec(T f1, T f2)
        {
          x=f1;
          y=f2;
    }

        vec()
         {  }

  vec operator+(const vec& v1)
 {
   vec result;
   result.x = v1.x+this->x;
   result.y = v1.y+this->y;
   return result;
 }
};

int main() {
  vec<int> v1(3,6);
  vec<int> v2(2,-2);
  vec<int> v3=v1+v2;


  vec<float> v4(3.9,6.7);
  vec<float> v5(2.0,-2.2);
  vec<float> v6=v4+v5;

}
```

Advantages of C++ Class Templates:

- One C++ Class Template can handle different types of parameters.
- 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.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

Function template

To perform identical operations for each type of data compactly and conveniently, we use function templates. we can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

Syntax:
The general form of a function template is

```
Template<class T>
returnType function_name (argument of type T)
{
//body of function with type T whenever appropriate
}
```

Using Template Functions: example

Using function templates is very easy: just use them like regular functions. When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max(int, int). Similarly the compiler generates definitions for max(char, char) and max(float, float) in this case.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements

template <class T>          //function template
T max(T a, T b)
{
        return a > b ? a : b ;
}

int main()
{

        cout << "max(10, 15) = " << max(10, 15) << endl ;
        cout << "max('k', 's') = " << max('k', 's') << endl ;
        cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
        return 0;
}
```

Output:
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2

Explanation:

The template keyword signals the compiler that I'm about to define a function template. The keyword class, within the angle brackets, might just as well be called type. As you've seen, you can define your own data types using classes, so there's really no distinction between types and classes. The variable following the keyword class (T in this example) is called the *template argument*.

Classes and Objects

One of the unique facilities provided by C language is "structure". Structures were used to group logically related data items. It was termed as a user defined data-type. Once the structure type was defined, we can create variables of that type.

Eg.

```
struct student
{
        char name[15];
        int roll_no;
        float total_marks;
};
```

For the above structure, we can define variables like

```
struct student s1,s2,s3;
```

But there are certain limitations in C structures.

- They don't allow data hiding. The structure members can be accessed using structure variable by any function anywhere in the scope.
- C does not allow the structure data type to be treated like built in data type.

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main precepts of OOP.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as private so that they can not be accessed directly by the external functions. In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword **struct** can be omitted in the declaration of structure variables.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structure for holding only data, and classes to hold both the data and functions.

The only difference between a structure and a class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.

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 data-type. So, classes are user-defined data types and behave like the built-in types of a programming language.

Classes allow the data and functions to be hidden, if necessary, from external use. Class, being a user-defined data type, we can create any number of variables for that class. These variables are called **Objects**.

Specifying a class

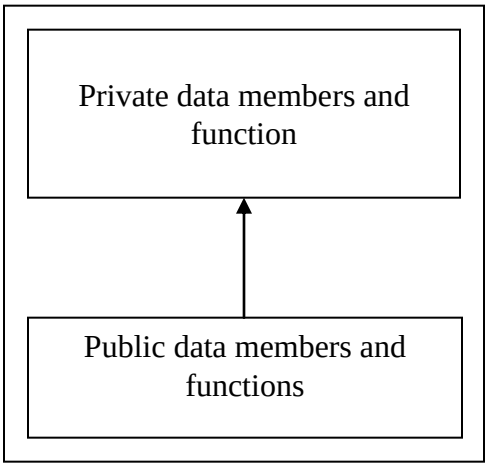
Specification of a class consists of two parts.

- class declaration
- function definition

Syntax:

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

};
```

- the class specification starts with a keyword "class" (like "struct" for structures), followed by a class-name. The class-name is an identifier.
- The body of the class is enclosed within braces and terminated by a semicolon.
- The functions and variables are collectively called class-members. The variables are, specially, called data members while the functions are called member functions.
- The two new keywords inside the above specification are – private and public. Those keywords are termed as **access-specifiers**, they are also termed as **visibility labels**. These are followed by colons.
    - The class members that have been declared as private can be accessed only from within the class. i.e., only the member functions can have access to the private data members and private functions.
    - On the other hand, public members can be accessed from anywhere outside the class (using object and dot operator).



    - All the class members are private by default. So, the keyword "private" is optional.

- If both the labels are missing, then, by default, all the members will be private and the class will be completely inaccessible by the outsiders (hence it won't be useful at all).

The key feature of OOP is data hiding. Generally, data within a class is made private and the functions are public. So, data will be safe from accidental manipulations, while the functions can be accessed from outside the class. However, it is not always necessary that the data must be private and functions public. In some cases, data may be public too and functions may be private.

Example of a class

```
class Test
{       int x,y;
        public:
        void get_data()
        {
            cin>>x>>y;
        }
        void put_data()
        {
           cout<<x<<y;
        }
        };
```

Creating Objects

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

```
Test t1; //memory for t1 is allocated.
```

The above statement creates a variable t1 of type Test. The class variables are known as **objects.** So, t1 is called object of class Test. We may also declare more than one object as follows.

```
Test t1, t2, t3;
```

When object is created, the necessary memory space is allocated to this object. The specification of class does not create any memory space for the objects. Objects can be created when a class is defined, as follows:

```
class Employee
{
        int id;
        char name[20];
public:
        void getname();
        void putname();
}e1,e2,e3;
```

The **object**s are also called **instance**s of the class. In terms of object, a class can be defined as a collection of objects (or instances) of similar type.

Accessing class members

The class members are accessed using a **dot operator**. But this works only for the public members. The dot operator is called **member access operator**.

The general format is

class-object.class-member ;

For public data members (if any), we can use following syntax to access them.

class-object.data-member ;

eg. obj1.data1 = obj1.data2 + obj1.data3;

In the above example, obj1 is an object of class. data1, data2, data3 are its public members.

For public functions, we can use following format to call them

class-object.function-name(argument-list);

eg. e1.getdata();

In the above example, e1 is an object of a class and getdata() is its member function. Here, no arguments have been listed since the function does not take any argument.

The private data and functions of a class can be accessed only through the member functions of that class.

Eg.

```
class A
{
        int x,y;
        void fu1();
public:
        int z;
        void fu2();
};

-   - - - - - - - - - -
-   - - - - - - - - - - -
A obj1;
obj1.x = 0; //generates error since x is private and can be accessed only thro' member functions
obj1.z = 0; //valid
obj1.fu1(); //generates error since fu1() is private
obj1.fu2();  //valid
```

Defining member functions (i.e. writing the body of a 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. (As has been done in the previous example of class Employee).

General form:

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

The function is, generally, defined immediately after the class-specifier. The function-name is proceeded by

- return-type of the function
- class-name to which the function belongs
- symbol with double colons(::). This symbol is called the scope resolution operator. This operator tells the compiler that the function belongs to the class class-name.

Eg.

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

In this example, the return-type is void which means the function "getdata()" doesn't return any value. The scope-resolution operator tells that the function "getdata()" is a member of the class "Employee". The argument list is also empty.

The member functions have some special characteristics:

- A program may have several different classes. These classes can use same function name. The scope-resolution operator will resolve which belongs to whom.
- Member functions can directly access private data of that class. A non-member function cannot do so. (Exception is friend function)
- A member function can call another member function directly, without using the dot operator.

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.

Eg.

```
class A
{
        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.

Eg.

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

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

Function definition with the 'inline' qualifier/keyword. This qualifier will make the function 'getdata()' an inline function

Nested Member Functions

An object of the class using dot operator generally, calls a member function of a class. 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".

Eg.

```
#include<iostream.h>

class Addition
{
        int a,b,sum;
        public:
        void read();
        void show();
        int add();
};
void Addition::read()
```

```
{
        cout<<"Enter a and b";
        cin>>a>>b;
}
void Addition::add()
{
        return(a+b);
}
void Addition::show()
{
        sum=add();    // nesting of function i.e member function called from another member function.
        cout<<endl<<"Sum of a and b is: "<<sum;
}

main()
{
        Addition a1;
        a1.read( );
        a1.show( );
        return(0);
}
```

The output of this program will be as follows:

```
        Enter a and b: 2
        3
        Sum of a and b is:5
```

Private Member Functions

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 it using dot operator.

Eg.

```
        class A
        {
                int a,b;
                void read(); //private member function

                public:
                void update();
                void write();
        };
        void A::update()
        {
                read(); //called from update() function. No object used.
```

```
}

- - - - - - - - - - - - -
- - - - - - - - - - - - - -
```

If a1 is an object of A, then the following statement is not valid.

```
a1.read();
```

This is because, read() is a private member function, hence cant be called using object and dot operator.

Memory Allocation for Objects

- As in the case of structures, memory space for objects is allocated when they are declared and not when the class is specified.
- But, we have to note one thing. The member functions are created and placed in the memory only once, when they are defined as a part of a class specification. All the objects belonging to the particular class will use the same member functions; no separate space is allocated for member functions when objects are created.
- However, the data members will hold different values for different object. So, space for member data is allocated separately for each object.

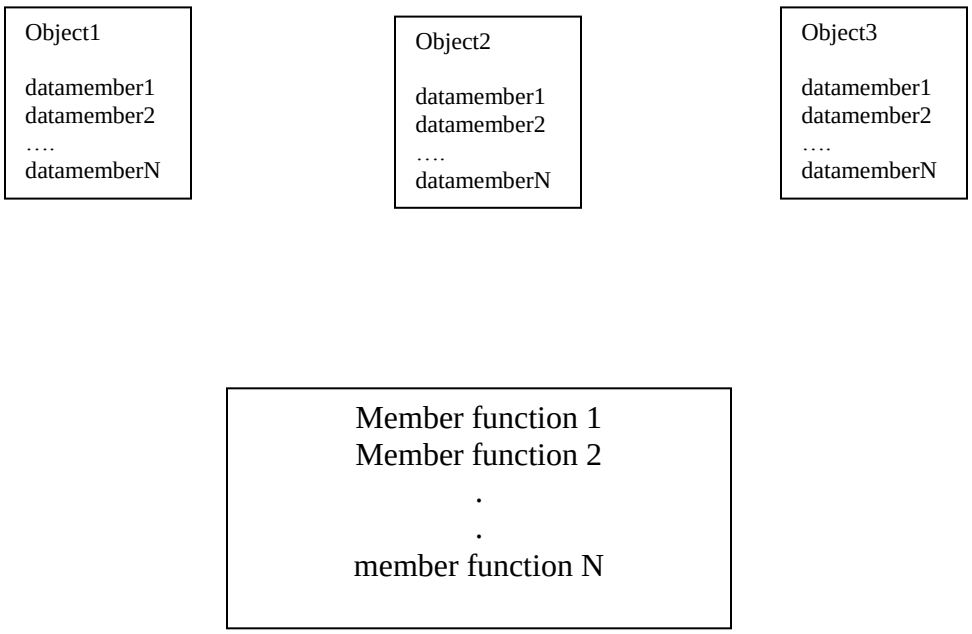


Fig: Objects in memory

Array of Objects

Array can be created of any data-type. Since a class is also a user defined data type, we can create array of objects (the variable of type class). Such an array is called an **array of objects**.

For eg. we had specified a class called Employee in some example above. If we have to keep records of 20 employees in an organization having two departments, then instead of creating 20 separate variables, we can create array of objects as follows.

```
Employee dept1[10];
Employee dept2[10];
```

The arrays dept1 and dept2 both has ten items (These items are objects of type Employee).

Since the array of objects is an array, we can access the members of the classes as follows:

```
dept1[0].getdata();
dept2[3].getdata();
```

Objects as function arguments

Like any other variable, objects can also be passed to the function, as an argument. There are two ways of doing this.

- pass by value
- pass by reference

In the "pass by value", the copy of the object is passed to the function. So, the changes made to the object inside the function do not affect the actual object. On the other hand, address of the object is passed in the case of pass by reference. So, changes made to the object inside the function are reflected in the actual object. The second method is considered more efficient.

Eg.

```
class Height
{
        int feet;
        int inches;
        public:
                void getHeight()
                {
                cout<<"Enter height in feet and inches";
                cin>>feet>>inches;
```

```
        }

        void putHeight()
        {
        cout<<"Feet:"<<feet;
        cout<<" and Inches:"<<inches;
        }

        void sum(Height,Height);
};

    void Height::sum(Height h1, Height h2)
    {
        inches = h1.inches + h2.inches;
        feet=inches/12;
        inches=inches%12;
        feet=feet + h1.feet + h2.feet;
    }

    int main()
    {
        Height h1,h2,h3;
        h1.getHeight();
        h2.getHeight();
        h3.sum(h1,h2);
        cout<<"Height 1:"<<h1.putHeight();
        cout<<"Height 2:"<<h2.putHeight();
        cout<<"Height 3:"<<h3.putHeight();
        return(0);
    }
```

In the above program, the function sum() takes two objects as arguments.

NOTE: A function can, not only, take objects as arguments, but they can also return object.

Eg.

In the above program if we modify the function sum() as follows:

```
        Height Height::sum(Height h1,Height h2) //function with return type Height
        {
            Height h3;
            h3.inches = h1.inches + h2.inches;
             h3.feet= h3.inches / 12;
            h3.inches= h3.inches % 12;
```

```
        h3.feet = h3.feet + h1.feet + h2.feet;
        return (h3); //object of type height returned
}
```

The above function will return an object h3 at the end of its execution.

Friend Function

A function is said to be a friend function of a class if it can access the members (including private members) of the class even if it is not the member function of this class.
In other words, a friend function is a non-member function that has access to the private members of the class.

Characteristics of a friend function:

- A friend function can be either global or a member of some other class.
- Since, friend function is not a part of the class, it can be declared anywhere in the public, private and protected section of the class.
- It cannot be called by using the object of the class since it is not in the scope of the class.
- It is called like a normal function without the help of any object.
- Unlike member functions, it cannot access member names directly. So, it has to use an object name and dot operator with each member name (like A.x)
- It, generally, takes objects as arguments.

A friend function is declared as follows:

```
class A
{
-- ---------- ----- 
-- ---------- ----- 
public:
        ----------
        ----------
friend void abc(void);          //friend function declaration
};
```

The declaration is preceded by the keyword “friend”. Its definition is written somewhere outside the class.
Eg.

```
class Avg
{
        int n1,n2;
        public:
         void getn()
        {
```

```
            cin>>n1>>n2;
        }
        friend int average(Avg a);
};
        //here "friend" is a keyword to specify that the function declared is a friend function
        //int is a return type of the friend function
        //average is the name of the friend function
        //Avg is the type of the argument type
        //a is an argument

int average(Avg a) //friend function definition
{
        return((a.n1 + a.n2)/2);
}

int main()
{
        Avg obj;
        obj.getn();
        cout<<"Mean:"<<average(obj);        //friend function called
        return(0);
}
```

Suppose we want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there is nothing like a friend function. Here is an example that shows how friend functions can act as a bridge between two classes. Read program carefully!!!!!!!!

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

class beta;   // needed for frenfunction declaration

class alpha
{
        private:
        int data;
        public:
        void get_data()
        {
                cin>>data;
        }
        friend int frenfunction(alpha, beta);  // friend function
};
```

```
class beta
{
        private:
        int data;
        public:
        void get_data()
        {
                cin>>data;
        }
        friend int frenfunction(alpha, beta);  // friend function
};

int frenfunction(alpha a,beta b)
{
        return(a.data+b.data);
}
main()
{
        alpha aa;
        beta bb;
        aa.get_data();
        bb.get_data();
        cout<<frenfunction(aa,bb)<<endl;
        getch();
        return 0;
}
```

Static Data Members

We have known that, each object of a class maintain their own copy of member data. In some cases, it may be necessary that all objects of a class have access to the same copy of a single variable. This can be made possible by using static variables.

- Only one copy of the static member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.
- It is initialized to zero when the first object of its class is created.
- Also known as class variable.

A static variable is declared using the “static” keyword.

```
class A
{
        static int count;
        int variable;
        public:
        A()
        {
                count++;
```

```
        }
        void get_var()
        {
                cin>>variable;
        }
        void put_var()
        {
                cout<<variable
        }
        void put_count()
        {
                cout<<count;
        }
};
int A::count;

main()
{
        A a,b,c;
        a.put_count();
        b.put_count();
        c.put_count();
        return(0);
}
```

The type and scope of each static member variable is defined outside the class definition. It is necessary since static data member are stored separately, rather than as a part of the object.

Also note, static variable can be assigned some initial value.
Eg.

```
int A::count=10;
```

will assign count the initial value 10

Output:

```
1 2 3
```

Static Member Function

In a class, functions can also be declared as static. Properties of static functions are

- they can access only other STATIC members (functions or variables) declared in the same class
- they can be called using class name
  eg. class_name::function_name

Example

```
class A
{
        int no;
        static int count;          //static member
        public:
        void set_no()
        {
```

```
            count++;
            no = count;
        }
        void put_no()
        {
            cout<<"No. is :"<<no;
        }
        static void put_count()          //static function accessing static member
        {
            cout<<endl<<"count:"<<count;
        }
};
int A::count;
main()
{
        A a1,a2;
        a1.set_no();
        a2.set_no();
        A::put_count();
        a1.set_no();
        a2.set_no();
        A::put_count();
        a1.put_no();
        a2.put_no();
        return(0);
}
```

Check output of this program.

Constructors and Destructors

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **input()** and **output()** to provide initial values to the private member variables. For example, the statement

t.input();

invokes the member function input(), which assigns the initial values to the data items of object **t**. Similarly, the statement

t.input(100,200);

passes the initial values as arguments to the function input(), where these values are assigned to the private variables of object **t**. All these function call statements are used with the appropriate objects that have already been created. These functions can not be used to initialize the member variables at the time of creation of their objects.

One of the aims of C++ is to create user-defined data types such as class that behave very similar to the built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

int x = 10;
float y = 56.67;

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. There are some features of class that enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the ***constructor*** which enables an object to initialize itself when it is created. This is known as ***automatic initialization*** of objects. It also provides another member function called the ***destructor*** that destroys the objects when they are no longer required.

Constructors
A constructor is a special member function whose task is to initialize the objects of its class. It has the name same as that of class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

- They are also used to allocate memory for a class object.
- They execute automatically when an object of a class is created.

- Constructor's name is same as that of class name.
- They should be declared in the "public" section.
- They do not have return types, not even void and therefore, and they cannot return any values.
- Like C++ functions, they can have default arguments.
- Constructor is NOT called when a pointer of a class is created.

There are, basically, three types of constructors.

- Default constructors
- Parameterized Constructors
- Copy Constructors

Default Constructors

A constructor that does not take any parameter is called default constructor. There are three possible situations for this.

4. If we do not provide any constructor with a class, the compiler provided one would be the default constructor. And it does not do anything other than allocating memory for the class object.

```
class A
{
    //no constructor
};
```

5. If we provide a constructor without any arguments then that is the default constructor.

```
class A
{
    A()
    {}
    //or
    A(void)
    {}

};
```

6. If we provide constructor with all default arguments, then that can also be considered as the default constructor.

```
class A
{
    A (int x=5)
    {}
};
```

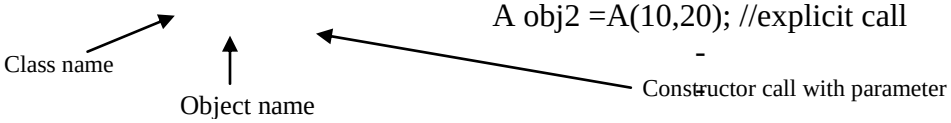
Parameterized Constructor

A constructor that takes arguments is called a parameterized constructor. Arguments are passed when the objects are created. This can be done in two ways.

- by calling the constructor explicitly
- by calling the constructor implicitly

For eg.

```
class A
{
                    int m,n;
                    public:
                            A(int x, int y); //parameterized constructor
                                                                    {
                     m=x;
        n=y;
        }
                      -                                     -       -
                      -                                     -       -

};

main()
{
                A obj1(10,20); //implicit call
               A obj2 =A(10,20); //explicit call
                      -                                     -       -
                      -                                     -       -
                      -                                     -       -
                      -                                     -       -
}
```



NOTE: Constructor functions can also be defined explicitly using scope resolution operator.

Copy Constructor

A copy constructor is called when an object is created by copying an existing object.
Eg.

```
A obj2(obj1);
```

The above statement would define the object obj2 and initialize it to the values of obj1.
The above statement can also be written as,

```
A obj2=obj1;
```

The process of initializing through a copy constructor is known as ***copy initialization***.
A copy constructor takes a reference to an object of the same class as itself (as an argument). ***We cannot pass the argument by value to a copy constructor***. When no copy

constructor is defined, the compiler supplies its own copy constructor. Remember the statement

obj2 = obj1;

will not invoke the copy constructor. However, if obj1 and obj2 are objects, this statement is legal and simply assigns the values of obj1 to obj2, member by member. This is the task of the overloaded assignment operator (=).

Eg.

```
class Data
{
                                        int info;
                                        public:
                                                                        Data()
        {}
                                                Data(int a)         // parameterized constructor
                                                                        {
                                                                                info
        }
        Data(Data &x)   // copy constructor
        {
                                        info=x.info;
        }
        void display()
        {
                                        cout<<info;
        }
};

int main()
{
                        Data d1(5); //parameterized constructor is called.
                        Data d2(d1); //copy constructor is called here.
                        Data d3=d2; //again a copy constructor is called here.
                                        Data d4;
                                d4 = d1;     // copy constructor not called
                                        cout<<"An info stored in d1";
                                                                        d1.display();
                                        cout<<"An info stored in d2";
                                                                        d2.display();

                                        cout<<"An info stored in d3";
                                                                        d3.display();
        cout<<"An info stored in d4";
```

```
d4.display();
return(0);
}
```

Constructors are called thrice in this program. The first one d1 calls the parameterized constructor, while the other two calls the copy constructor

The output will be

```
An info stored in d1 5
An info stored in d2 5
An info stored in d3 5
An info stored in d4 5
```

Constructor Overloading

The process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor is defined in a class, it is called constructor overloading.

In the above example of class Data, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply one argument, while the third one gets invoked when an object is passed as an argument.

Eg.

```
Data obj1;
```

This statement would automatically invoke the first one.

```
Data obj2(5); invokes 2nd constructor
Data obj3(obj2); invokes 3rd constructor
```

Constructor with default argument

Like functions, constructors can also have default arguments.

Eg.

```
class A
{
        int a,b,c;
        pulic:
                A()
                {
                        a=0
                }
                A(int x,int y=10,int z=20)
                {
                        a=x;b=y;c=z;
                }
                void display()
                {
                        cout<<a<<b<<c;
                }
};
```

```
main()
{
                                        A obj1;
                                    A obj2(5,10,15);
                                        A obj3(6);
                                      obj1.display();
                                      obj2.display();
                                      obj3.display();
                                        return(0);
}
```

Output:

```
0 0 0
5 10 15
6 10 20
```

In this program, there are two constructors. The second one has default parameters. Such a constructor is called a constructor with default argument. These constructors have all the properties of the **functions with default argument.**

It is important to distinguish between the default constructor A::A() and the default argument constructor A::A (int x = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A   a;
```

The ambiguity is whether to call A::A() or A::A(int x = 0).

Example

```
#include<iostream.h>
#include<conio.h>
class test
{
  int x,y;
  public:
  //test()   // No need to use this default constructor
  //{}
  test(int p=10,int q = 20) // when this constructor is called with no arguments, it
  {                         // becomes default constructor
    x = p;
    y = q;
  }
  void display()
  {
```

```
    cout<<"X = "<<x<<"\n"<<"Y = "<<y<<endl;
  }
};

void main()
{
  int x,y;
  cin>>x>>y;
  test t(x,y);  // call to default argument constructor
  t.display();
  test t1;       //  No argument is passed, so becomes default constructor
  t1.display();
  getch();
}
```

Dynamic initialization of objects

Class objects can be initialized dynamically (i.e. at the run time). The users provide the values at the run time.

Advantage: various initialization formats can be provided using constructor overloading.

Eg.

```
class Area
{
int l,b,area;
  public:
        Area()
        {
                l=1;
                b=1;
                area=1;
        }
        Area(int a, int br)
        {
                l=a;
                b=br;
                area=l*b;
        }
        Area(int a)
        {
                l=a;
                b=0;
                area=l*l;
        }
};
```

```
void main()
{
        Area a1,a2;
        int l,b,len;
        cin>>l>>b;
        a1=Area(l,b);
        cin>>len;
        a2=Area(len);
        - - - - - - - - -
        - - -- - - - - - -
}
```

NOTE: In dynamic initialization, we are explicitly calling the required constructor. And we haven't used the object and dot operator.

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 (~).
Eg.

A destructor for the class Area will look like

```
~Area()
{ - - - - - - -
  - - - - - -
}
```

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.

```
#include<iostream.h>
#include<conio.h>
class A
{
        static int count;
        public:
        A()
        {
            count++;
            cout<<count<<endl;
        }
        ~A()
        {
            cout<<count<<endl;
            count--;
        }
};

int A::count;

main()
{
      A a1,a2,a3;
      getch();
      return 0;
}
```

File I/O Operations in C++

Working with Files and File I/O operations

A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files. A program typically involves either or both of the following kinds of data communication:

3. Data transfer between the console unit and the program. (already discussed)
4. Data transfer between the program and a disk file.

The I/O system of C++ handles file operations which are very much similar to the console I/O operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as **input stream** and the one that receives data from the program is known as **output stream**. In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file.

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

The I/O system of C++ contains a set of classes that define the file handling methods. These include **ifstream**, **ofstream** and **fstream**. These classes are derived from fstreambase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in **fstream** and therefore we must include this file in any program that uses files.

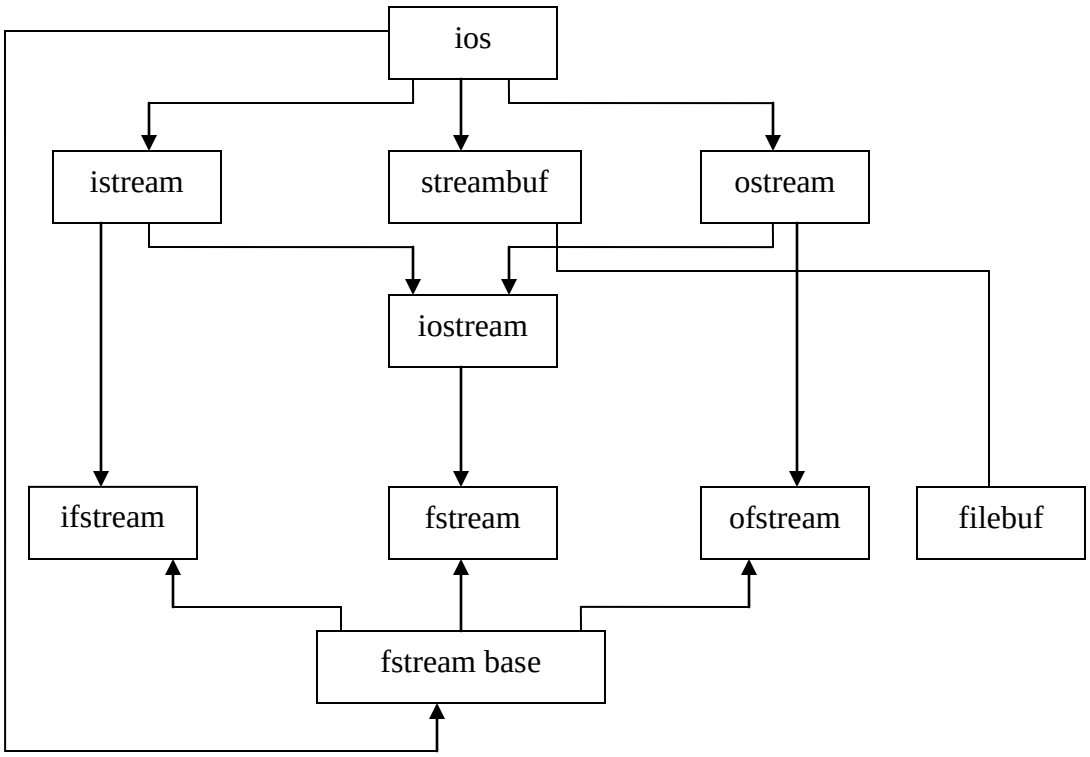


Fig. stream classes for file I/O operations

ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.

Opening and Closing a File
We must first create a file stream and then link it to the file name. A file stream can be defined using the classes **ifstream**, **ofstream**, and **fstream**. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways

3. Using the constructor function of the class
4. Using the member function **open()** of the class

Opening Files Using Constructor
Opening a file using constructor involves the following steps:

3. Create a file stream object to mange the stream using the appropriate class. That is to say, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
4. Initialize the file object with the desired filename.

For example: ofstream file("student"); //write only
ifstream file("employee"); // read only

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
        char name[20];
        int age;
        char sec;
        ofstream ofile("Student.txt");
        cout<<"\nEnter name, age section: ";
        cin>>name>>age>>sec;
        ofile<<name<<endl;
        ofile<<age<<endl;
        ofile<<sec;
        ofile.close();
        ifstream ifile("student.txt");
        ifile>>name;
        ifile>>age;
        ifile>>sec;
        cout<<name<<"\t"<<age<<"\t"<<sec;
        ifile.close();
        getch();
}
```

When a file is opened for writing only, a new file is created if there is no file of that name. If a file by that name exists already, then its contents are deleted and the file is presented as a new one.

Opening Files Using open()

The function open() can be used to open multiple files that use the same stream object. In such cases, we may create a single stream object and use it to open each file in turn. This is done as

```
        file-stream-class      stream-object;
        stream-object.open(“filename”);
```

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
main()
{
        char name[20];
        int age,salary;
        char sec;
        ofstream ofile;
        ofile.open("student.txt");
```

```
        cout<<"\nEnter name, age section: ";
        cin>>name>>age>>sec;
        ofile<<name<<endl;
        ofile<<age<<endl;
        ofile<<sec;
        ofile.close();
        ofile.open("teacher.txt");
        cout<<"\nEnter name and salary of teacher: ";
        cin>>name>>salary;
        ofile<<name<<endl;
        ofile<<salary;
        ofile.close();
        getch();
}
```

File Modes with open()

The open function can take two arguments, the second one for specifying the file mode. The general form of the function open with two arguments is

```
stream-object.open("filename", mode);
```

The second argument (called file mode parameter) specifies the purpose for which the file is opened.

Parameter	**Meaning**
ios :: app	append to end of file
ios :: ate	go to end of file on opening
ios :: in	open file for reading only
ios :: out	open file for writing only
ios :: binary	binary file
ios :: nocreate	open fails if the file does not exists
ios :: noreplace	open fails if the file already exists
ios :: trunc	delete the contents of the file if it exists

The mode can combine two or more parameters using the bitwise OR operator (|).

put() and get() Functions

The function put() writes a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        char ch;
```

```
        fstream file;
        file.open("myfile.txt",ios::out); // write only
        cin.get(ch);
        while(ch != '\n')
        {
                file.put(ch);
                cin.get(ch);
        }
        file.close();
        file.open("myfile.txt",ios::in);  // read only
        while(file)
        {
                file.get(ch);
                cout<<ch;
        }
        file.close();
        getch();
}
```

write() and read() Functions

The functions write() and read(), unlike the functions put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. For example- an **int** takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit **int** will take four bytes to store it in the character form. **write()** and **read()** functions take two arguments. The first is the address of the variable, and the second is the length of that variable in bytes. The address of the variable must be cast to type char* (pointer to character type).

```
//writing array of integers using write()
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        int x[] = {100,200,300,400};
        int i;
        fstream file;
        file.open("myfile.dat",ios::out|ios::binary);
        file.write((char*)&x,sizeof(x));
        file.close();
        for(i = 0;i < 4;i++)
        x[i] = 0;
        file.open("myfile.dat",ios::in|ios::binary);
        file.read((char*)&x,sizeof(x));
        for(i = 0;i < 4;i++)
        cout<<x[i];
```

```
	getch();
}
```

Writing and Reading Class Objects (Object I/O)

When writing an object we generally want to use binary mode. This writes the same bit configuration to disk that was stored in memory. The binary input output functions **write()** and **read()** are used for this.

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class person
{
	protected:
	char name[20];
	int age;
	public:
	void getData()
	{
		cin>>name>>age;
	}
	void showData()
	{
		cout<<name<<"\t"<<age<<endl;
	}
};

main()
{
	person p,q;
	fstream file;
	file.open("person.dat",ios::out|ios::binary);
	cout<<"\nEnter person's data: ";
	p.getData();
	file.write((char *)(&p),sizeof(p));
	file.close();
	file.open("person.dat",ios::in|ios::binary);
	file.read((char *)(&q),sizeof(q));
	q.showData();
	getch();
	return 0;
}
```

Writing Multiple Objects

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class person
{
        protected:
        char name[20];
        int age;
        public:
        void getData()
        {
                cin>>name>>age;
        }
        void showData()
        {
                cout<<name<<"\t"<<age<<endl;
        }
};
main()
{
        char ch;
        person p;
        fstream file;
        file.open("person.dat",ios::out|ios::in|ios::binary);
        do
        {
                cout<<"\nEnter person's data: ";
                p.getData();
                file.write((char *)(&p),sizeof(p));
                cout<<"Enter another person? ";
                cin>>ch;
        }while(ch == 'y');
        file.seekg(0);
        file.read((char *)(&p),sizeof(p));
        while(!file.eof())
        {
                p.showData();
                file.read((char *)(&p),sizeof(p));
        }
        cout<<endl;
        getch();
        return 0;
}
```

File Pointers

Each file object has associated with it two integer values called the *get pointer* and the *put pointer*. These are also called the current get position and the current put position, or simply the current position. These values specify the byte number in the file where writing or reading will take place. The **seekg()** and **tellg()** functions allow us to set and examine the get pointer, and the **seekp()** and **tellp()** functions perform these same actions on the put pointer.

Specifying the offset
We can move the file pointer to a desired location using the **seek** function. The argument to these functions represents the absolute position in the file. Syntax for this is:

```
seekg(offset, refposition);
seekp(offset, refposition);
```

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition takes one of the following three constants defined in the class **ios**

ios :: beg	start of file
ios :: cur	current position of the pointer
ios :: end	end of the file.

The **seekg()** function moves the associated file's *get pointer* while the **seekp()** function moves the associated file's *put pointer*. If **file** is an object of ofstream, then some some sample pointer offset calls and their actions are:

Seek call	**Action**
file.seekg(0,ios::beg);	go to start
file.seekg(0,ios::cur);	stay at the current position
file.seekg(0,ios::end);	go to the end of file.
file.seekg(m,ios::beg);	move to (m+1)th byte in the file
file.seekg(m,ios::cur);	go forward by m byte from the current position

```
// program using file pointers
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class person
{
        protected:
        char name[20];
        int age;
        public:
        void getData()
```

```
	{
		cin>>name>>age;
	}
	void showData()
	{
		cout<<name<<"\t"<<age<<endl;
	}
};
main()
{
	person p;
	fstream file;
	file.open("group.dat",ios::in|ios::binary);
	file.seekg(0,ios::end);
	int endposition = file.tellg();
	int n = endposition/sizeof(p);
	cout<<"\nNumber of records in file = "<<n;
	cout<<"\nEnter person number: ";
	cin>>n;
	int pos = (n-1)*sizeof(p);
	file.seekg(pos);
	file.read((char*)&p,sizeof(p));
	p.showData();
	cout<<"\nRecords in file are:\n";
	file.seekg(0,ios::beg);
	file.read((char*)&p,sizeof(p));
	while(!file.eof())
	{
		p.showData();
		file.read((char*)&p,sizeof(p));
	}
	file.close();
	getch();
	return 0;
}


// File copy program. This program copies the content of one file into another.
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#include<ctype.h>
void main()
{
	char ch;
	fstream file1,file2;
```

```
        file1.open("test1.txt",ios::out|ios::in);
        file1<<"welcome to object oriented programming";
        file1.close();
        file1.open("test1.txt",ios::in);
        file2.open("test2.txt",ios::out);
        while(file1)
        {
                file1.get(ch);
                ch = toupper(ch);
                file2.put(ch);
        }
        file1.close();
        file2.close();
        file2.open("test2.txt",ios::in);
        while(file2)
        {
                file2.get(ch);
                cout<<ch;
        }
        file2.close();
        getch();
}
```

Error Handling During File Operations

The class **ios** supports several member functions that can be used for handling error occurred during file operations. Some functions are

eof() returns true if end of file is encountered while reading, otherwise return false.

fail() returns true when an input or output operation has failed.

bad() returns true if an invalid operation is attempted or any unrecoverable error has occurred.

good() returns true if no error has occurred.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
        char ch;
        fstream file;
        file.open("myfile.txt",ios::in);  // read only
        if(file.fail())
        {
                cout<<"\nCould not open the file";
```

```
	}
	else
	{
		while(file)
		{
			file.get(ch);
			cout<<ch;
		}
		file.close();
	}
	getch();
}
```

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
	char ch;
	fstream file;
	file.open("myfile.txt",ios::in);  // read only
	if(!file.good())
	{
		cout<<"\nCould not open the file";
	}
	else
	{
		while(file)
		{
			file.get(ch);
			cout<<ch;
		}
		file.close();
	}
	getch();
}
```

Functions

A function is a single comprehensive unit that performs a specified task. This specified task is repeated each time the function is called. Functions break large programs into smaller tasks. They increase the modularity of the programs and reduce code redundancy. Like in C, C++ programs also should contain a main function, where the program always begins execution. The main function may call other functions, which in turn will again call other functions.
When a function is called, control is transferred to the first statement of the function body. Once the statements of the function get executed (when the last closing bracket is encountered) the program control return to the place from where this function was called.

Function Prototype (Function declaration)
Function prototype lets the compiler know the structure of function in terms of its name, number and type of arguments and its return type.

Syntax:

return-type function-name(datatype1, datatype2, …,datatype n);

Function Call
Function call is the process of making use of function by providing it with the parameters it needs. We call a function as follows.

function-name (argument1, argument2, .. ,argument n);

Function Definition
Function definition is a process of defining how it does what it does or in other words, during function definition, we list the series of codes that carry out the task of the function. A function is defined as follows,

```
return-type function-name(datatype1 variable1, datatype2 var2, …., datatype n var n)
{
        …… ;
        ……; //body of the function
        ……;
}
```

Default Arguments

In C++, a function can be called without specifying all its arguments. But it does not work on any general function. The function declaration must provide default values for those arguments that are not specified. When the arguments are missing from function call, default value will be used for calculation.

```
#include<iostream.h>
float interest(int p, int t = 5, float r = 5.0);
main()
```

```
{
        float rate, i1,i2,i3;
        int pr , yr;
        cout<<"Enter principal, rate and year";
        cin>>pr>>rate>>yr;
        i1=interest(pr ,yr ,rate);
        i2=interest(pr , yr);
        i3=interest(pr);
        cout<<i1<<i2<<i3;
        return(0);
}

float interest(int p, int t, float r)
{
        return((p*t*r)/100);
}
```

In the above program, t and r has default arguments. If we give, as input, values for pr, rate and yr as 5000, 10 and 2, the output will be

1000 500 1250

NOTE: The default arguments are specified in function declaration only and not in function definition.
Only the trailing arguments can have default values. We must add defaults from right to left. We cannot provide a default value to a particular argument at the middle of an argument list. Default arguments are used in the situation where some arguments have same value. For eg., interest rate in a bank remains same for all customers for certain time.

Inline Functions

We say that using function s in a program is to save some memory space because all the calls to the functions cause the same code to be executed. However, every time a function is called, it takes a lot of extra time in executing a series of instructions. Since the tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function are carried out when a function is called. When a function is small, considerable amount of time is spent in such overheads.

C++ has a solution for this problem. To eliminate the cost of calls to small functions, C++ proposed a new feature called INLINE function.
When a function is defined as inline, compiler copies it s body where the function call is made, instead of transferring control to that function.
A function is made inline by using a keyword "inline" before the function definition.

Eg.

```
        inline void calculate_area(int l,int b)
        {
```

```
        return(l * b);
    }
```

It should be noted that, the inline keyword merely sends request, not a command, to a compiler. The compiler may not always accept this request. Some situations where inline expansion may not work are

- for functions having loop, switch or goto statements
- for recursive functions
- functions with static variables
- for functions not returning values, if a return statement exists

Inline functions must be defined before they are called.

Eg.

```
#include<iostream.h>

inline float lbtokg(float lbs)
{
    return (0.453 * lbs);
}


main()
{
    float lbs, kgs;
    cout<<"Enter weight in lbs:";
    cin>>lbs;
    kgs=lbtokg(lbs);
    cout<<"Weight in kg is "<<kgs;
    return (0);
}
```

Exercise:

When do we use inline function? Explain with example.
When do we use default argument? Explain with example.

Function Overloading

Function that share the same name are said to be **overloaded functions** and the process is referred to as **function overloading**. i.e. function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters, or different sequence of parameters or different number of parameters. The number, type or sequence of parameters for a function is called the function signature. When we call the function, appropriate function is called based on the parameter passed. Two functions differing only in their return type can not be overloaded. For eg-
int add(int , int) and float add(int, int)

A function call first matches the declaration having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection will involve the following steps:

- the compiler first tries to find an exact match in which the types of actual arguments are the same and uses that function
- if an exact match is not found, the compiler uses the integral promotion to the actual parameters, such as,
    - char to int
    - float to double to find the match
- If both of the above fail, the compiler tries to use the built-in conversions and then uses the function whose match is unique.

```
#include<iostream.h>
//function declaration
float perimeter(float);
int perimeter(int,int);
int perimeter(int,int,int);

main()
{
        cout<<"Perimeter of a circle: "<<perimeter(2.0)<<endl;
        cout<<"Perimeter of a rectangle: "<<perimeter(10,10)<<endl;
        cout<<"Perimeter of a triangle: "<<perimeter(5,10,15);
        return (0);
}

//function definition
float perimeter(float r)
{
        return(2*3.14*r);
}
int perimeter(int l,int b)
{
        return(2*(l+b));
}
int perimeter(int a,int b,int c)
```

```
{
        return(a+b+c);
}
```

In the above program, a function "perimeter" has been overloaded. The output will be as follows:

Perimeter of a circle 12.56
Perimeter of a rectangle 40
Perimeter of a triangle 30

Recursive function

Recursion is a powerful technique of writing complex algoriths in an easy way. It defines the problem in terms of itself. In this technique , a large problem is divided into smaller problem of similar nature as original problem.so that the smaller problem is easy to solve and in the most case they can be solved easily. Hence to implement this techinique, a programming language support the function that is capable of calling itself. C++ support such function and these function are called recursive functions

For example: to find the factorial of a given number

```
Int mani()
{
        int num;
        cout<<"Enter a number";
        cin>>num;
        int f=fact(num);
        cout>>"the factorial of given number is"<<f;
        getch();
        return 0;

}

int fact(int num)
{
        If(num==0)
                Return 1;
        Else
                Return (num*fact(num-1));
}
```

Inheritance

Inheritance is the most powerful feature of object-oriented programming. Inheritance is the process of creating new classes, called derived classes, from existing classes or base classes. The class inherits all the capabilities of the base class but can add refinements of its own.

When new class is created based on some other class using inheritance, the newly created class is called the derived class or sub-class, while the class on which it is based is called base class or superclass.

Inheritance is also called a 'kind of relationship'. Inheritance supports the concept of '**reusability**'. Once a class has been written and tested, its features can be adapted by other programmers whenever required.

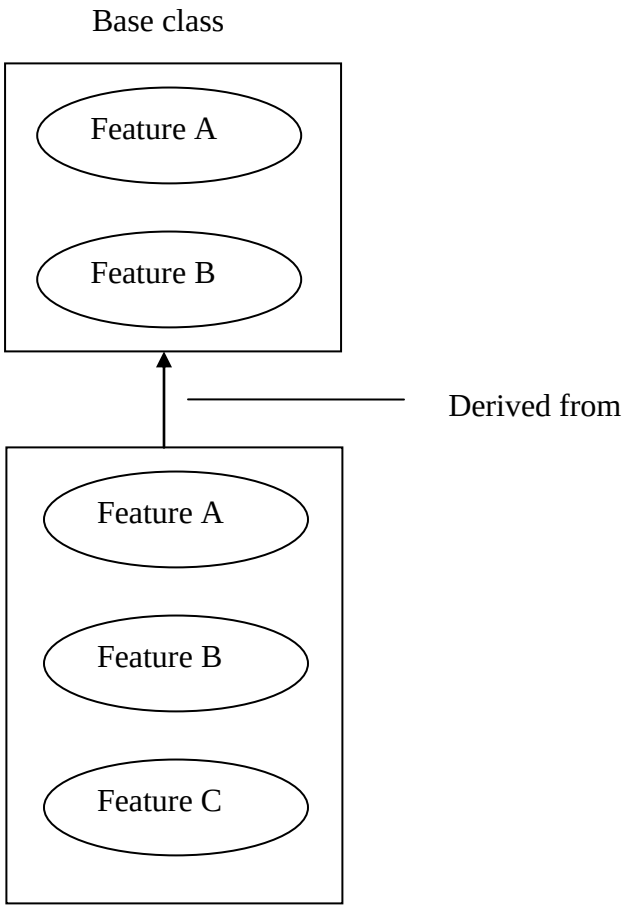
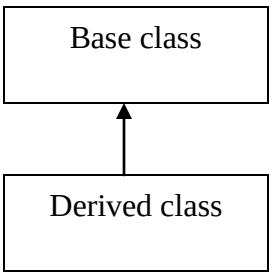


Fig. Inheritance

Types

There are 5 types of inheritance

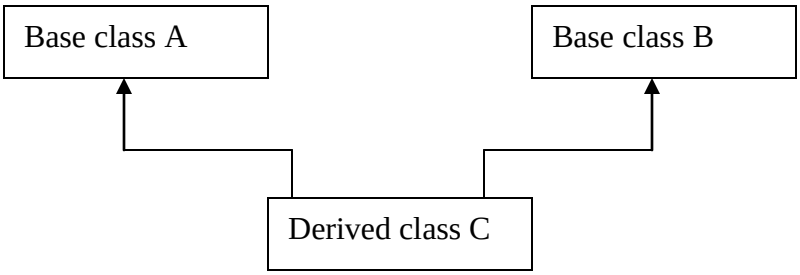
1) Single Inheritance – when a class is derived from only one base class, then it is called single inheritance. We can represent it as



Eg-

```
class A       // base class
{
        private:
        {
                ….  // body part
                …
        }
        public:
        {
                …..// body part
                …..
        }
};
class B : public A    // derived class B
{
        ……
        …..
};
```

2) Multiple Inheritance: When a class is derived from two or more base classes, it is called multiple inheritance. It can be represented as,

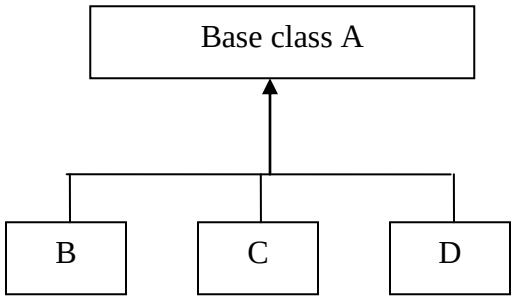


Eg-

```
class A
{
  …….
   ……
};

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

Here two base classes A and B have been created and a derived class C is created from both base classes.

3) Hierarchical Inheritance: When two or more than two classes are derived from one base class, it is called hierarchical inheritance. It can be represented as

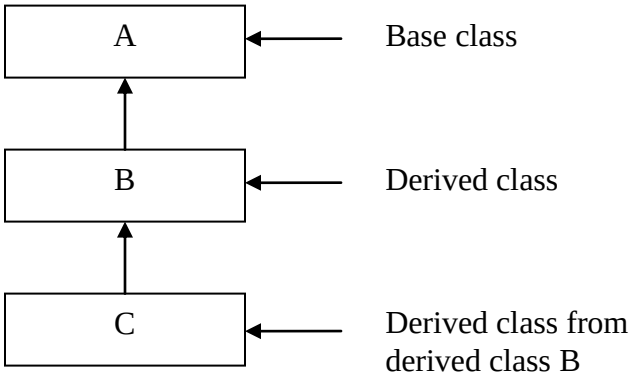


Eg-

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

……
};
Here two classes B and C are derived from same base class A.

4) Multilevel Inheritance: The mechanism for deriving a class from another derived class is known as multilevel inheritance and can be presented as

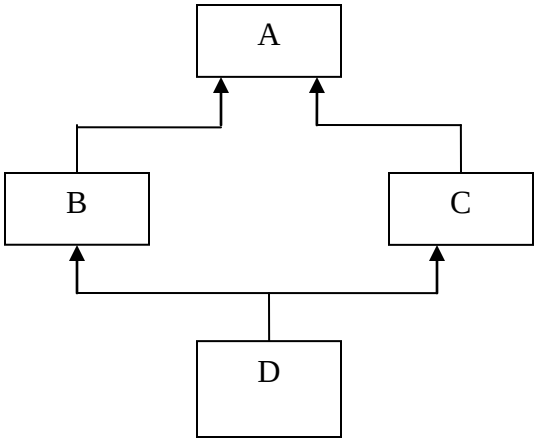


E.g.-

```
Class A
{
  …..
  …..
};
classs B : public A
{
  …….
  ……
};
class C : public B
{
  …..
  …..
};
```

In this example class B is derived from base class A and class C is derived from derived class B.

5) Hybrid Inheritance: This inheritance is the combination of multiple and hierarchical inheritance, it can be represented as



E.g-

```
class student
{
      ……
      …..
};
class test : public student
{
    …….
    …….
};
class sport : public student
{
    ……
    …...
};
class result : public test, public sport
{
   ……
   ….
};
```

In the example, there is one base class student. The class test and sport are derived from student and class result is derived from both test and sport classes. This shows the combination of multiple and hierarchical inheritance.

Sub-class Definition

A subclass can be defined by specifying its relationship with base class along with its own details. General form of defining sub class is

```
class    subclass-name : visibility-mode     baseclass-name
{
  ……….
  ……
};
```

The colon indicates that the inheritance has been used. i.e. the sub class has been derived from base class. The visibility mode specifies whether the features of the base class are derived privately, publicly or protectedly. This is optional and if absent, the default is private.
The three visibility modes are

- public
- private
- protected

When the base class is ***publicly inherited***, all the public members of the base class become public members of the derived class. So, they can be accessed through the objects of the derived class. All the protected members become protected members of the derived class.

When the base class is ***privately inherited***, all the public and protected members of the base class become private members of the derived class. So, these can not be accessed outside the class directly through the derived class object, but might be accessed through public functions in the derived class. Like general private members, these members can be used freely within the derived class.

NOTE: private members can not be inherited at all. Only the public and protected ones can be inherited.

When a member is defined with ***protected*** access specifier, these members can be accessed from that class and also from the derived class of this base class. But can not be accessed from any other function or class. i.e. protected members act as public for derived class and private for other classes.

When base class is derived using ***protected*** mode, all the protected and public members of the base class become protected members of the derived class. This means, like a private inheritance, these members can not be directly accessed through object of the derived class. But can be used freely within the derived class. Whereas, unlike a private inheritance, they can still be inherited and accessed by subsequent derived classes. In other words, protected inheritance does not end a hierarchy of classes, as private inheritance does.

Overriding base class members
In C++, a base class member can be overridden by defining a derived class member with the same name as that of the base class member. Consider the program

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

```
class aclass
{
      public:
         void disp(void)
          {
              cout<<"Base"<<endl;
          }
};
class bclass : public aclass
{
      public:
         void disp(void)
          {
               cout<<"Derived"<<endl;
           }
};
void main()
{
      bclass Bvar;
      Bvar.disp();
}
```

Here, the function disp() is overridden.

- If the function is invoked from an object of the derived class, then the function in the derived is executed.
- If the function is invoked from an object of the base class, then the base class member function is invoked.

Ambiguities in Multiple Inheritance

When a class inherits from multiple base classes, a whole part of ambiguities creep in. for eg- what happens when two base classes contain a function of the same name? Consider the following program,

```
#include<iostream.h>
class base1
{
    public:
        void disp(void)
         {
             cout<<"Base1"<<endl;
          }
};
class base2
{
    public:
```

```
        void disp(void)
         {
               cout<<"Base2"<<endl;
          }
};
class derived : public base1, public base2
{
   // Empty
};
void main()
{
   derived Dvar;
    Dvar.disp();   // Ambiguous
}
```

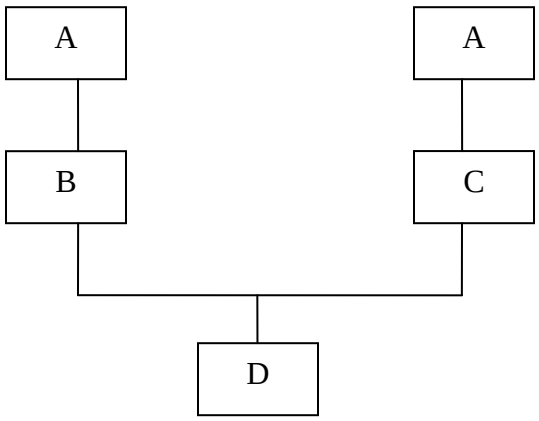
Here, the reference to disp() is ambiguous because the compiler does not know whether disp() refers to the member in class base1 or base2. This ambiguity can be resolved using the scope resolution operator as

```
void main()
{
     derived Dvar;
     Dvar.base1::disp();
     Dvar.base2::disp();
}
```

This ambiguity can also be resolved by overriding. That is, the members can be redefined in the derived class. For eg-

```
class base1
{
  ……
  …...
};
class derived : public base1, public base2
{
     public:
         void disp(void)
         {
             base1::disp();
             base2::disp();
             cout<<"Derived class"<<endl;
           }
};
void main()
{
        derived Dvar;
         Dvar.disp();     // Not ambiguous
```

}
Another ambiguity that arises in multiple inheritance is the possibility of the derived class having multiple copies of the same base class. Consider the following diagram



Inheritance from a base class via different paths

In the above figure, class D inherits from two base classes namely B and C. B and C in turn are derived from class A. As a result, class D would have two copies of class A. There are two ways of resolving this ambiguity. One way is to the scope resolution operator and the other way is to use virtual base classes.

Containership
When a class contains object of another class as its member data, it is termed as containership. The class which contains the object is called container class. Containership is also termed as "class within class".

```
class A
{
   ……
   ……
};
class B
{
  …..
  A obj1;
  …..
};
```

Here, class B contains object of class A. So B is the container class.

'containership' is also called **'has-a'** relationship.

In some situations, inheritance and containership relationship serves similar purpose. Containership is useful with classes that act like a data type. The object of these classes can be used almost like other variables in the class.

In inheritance, if a class **B** is derived from a class **A**, then "**B** is a kind of **A**". This is because B has all the characteristics of A, and in addition some of its own. So, inheritance is often called a **"kind of"** relationship.

```
class Manager
{
  char name[20];
  int age;
  public:
  void getdata()
  {
      cin>>name>>age;
  }
  void putdata()
  {
      cout<<name<<age;
  }
};
class Employee    // Employee is container class
{
  char department[20];
  Manager m;    // Object of class Manager
  public:
  void getdata()
  {
    m.getdata();
    cin>>department;
  }
  void putdata()
  {
    m.putdata();
    cout<<department;
  }
};
void main()
{
   Employee e;
   e.getdata();
   e.putdata();
   getch();
}
```

Abstract Class

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class. It is a design concept in program development and provides a base upon which other classes may be built.

Virtual Base Class

The principle behind the virtual base class is very simple. When the same class is inherited more than once via multiple paths, multiple copies of the base class members are created in memory. By declaring the base class inheritance as virtual, only one copy of the base class is inherited. A base class inheritance can be specified as a virtual using the virtual qualifier.

```
class A
{
  …..
  …..
};
class B1 : virtual public A
{
  ….
  ….
};
class B2 : public virtual A
{
  ……
  …...
};
class C : public B1, public B2
{
  ……
  ……
};
```

Keywords 'virtual' and 'public' can be used in either way.

When a class is made virtual base class, only one copy of that is inherited, regardless of how many number of inheritance path exists between the virtual base class and derived class. Since there is only one copy, there is no ambiguity.

Constructors in Derived Class

One important thing to note is that as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. While applying inheritance, we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When

both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them onto the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor. Example-

```
#include<iostream.h>
class alpha
{
  int x;
  public:
  alpha(int p)
  {
    x = p;
    cout<<"\nAlpha initialized";
  }
  void show_x()
  {
    cout<<"\nX = "<<x;
  }
};
class beta
{
  float y;
  public:
  beta(float q)
  {
      y = q;
      cout<<"\nBeta initialized";
  }
  void show_y()
  {
     cout<<"\nY = "<<y;
  }
};
class gamma : public alpha, public beta
{
  int m,n;
  public:
```

```
  gamma(int a,float b,int c,int d): beta(b),alpha(a)
  {
    m = c;
    n = d;
    cout<<"\nGamma initialized";
  }
  void show_mn()
 {
     cout<<"\nM = "<<m<<"\nN = "<<n;
 }
};
void main()
{
  gamma g(10,5.5,50,60);
  g.show_x();
  g.show_y();
  g.show_mn();

}
```

Constructor and destructors in derived class: order of executions

Situation remains understandable until both the base and its derived class have Constructors and/or Destructors. Since the derived class contains more than one Constructors and/or Destructors, it becomes confusing which one will be called when.

This is because when an object the inherited class is constructed both the constructors (base's and its own) should be invoked and same applies when it gets destructed.

This article will clear all this!

Consider the following example program:

```
// -- INHERITANCE --
// Constructors, Destructors
// and Inheritance
#include<iostream.h>

// base class
class base
{
public:
  base(){cout<<"Constructing Base\n";}
  ~base(){cout<<"Destructing Base\n";}
```

```
};

// derived class
class derived:public base
{
public:
  derived(){cout<<"Constructing Derived\n";}
  ~derived(){cout<<"Destructing Derived\n";}
};

void main(void)
{
  derived obj;

  // do nothing else, only
  // construct and destruct
  // the inherited class object
}
```

OUTPUT:

```
Constructing Base
Constructing Derived
Destructing Derived
Destructing Base
Press any key to continue
```

So here is the general rule:

Constructors are called in the order of derivation and Destructors in the reverse order.

One more example will clear the confusions, if any.:

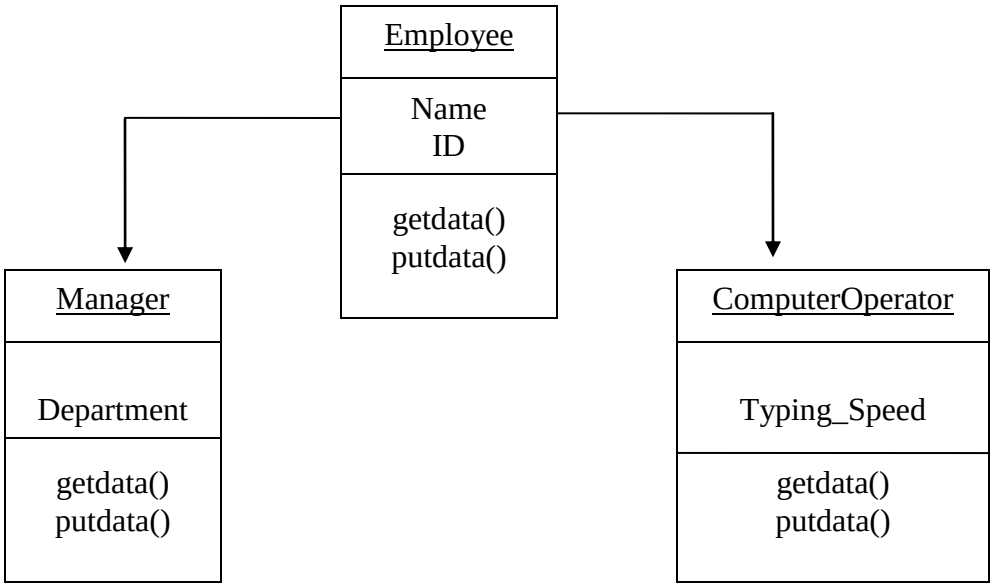
```
// -- INHERITANCE --
// Constructors, Destructors
// and Inheritance
#include<iostream.h>

// base class (1)
class base
{
public:
  base(){cout<<"Constructing Base\n";}
  ~base(){cout<<"Destructing Base\n";}
};
```

```
// derived class
// derived form 'base'
class derived1:public base
{
public:
  derived1(){cout<<"Constructing Derived1\n";}
  ~derived1(){cout<<"Destructing Derived1\n";}
};

// derived from a derived class
// 'derived1'
class derived2:public derived1
{
public:
  derived2(){cout<<"Constructing Derived2\n";}
  ~derived2(){cout<<"Destructing Derived2\n";}
};

void main(void)
{
  derived2 obj;

  // do nothing else, only
  // construct and destruct
  // the inherited class object
}
```

OUTPUT:

```
Constructing Base
Constructing Derived1
Constructing Derived2
Destructing Derived2
Destructing Derived1
Destructing Base
```

Lab Sheet - 3



7. Write a C++ program to represent the above inheritance scheme. Also write a main() function to test the classes, Manager and ComputerOperator, by creating their objects, taking input and displaying the corresponding values.

8. Imagine a college hires some lecturers. Some lecturers are paid in period basis, while others are paid in month basis. Create a class called **lecturer** that stores the **ID**, and the **name of lecturers**. From this class derive two classes: **PartTime**, which adds payperhr (type float); and **FullTime**, which adds paypermonth (type float). Each of these three classes should have a **readdata()** function to get its data from the user, and a **printdata()** function to display its data.

   Write a **main()** program to test the **FullTime** and **PartTime** classes by creating instances of them, asking the user to fill in their data with **readdata()**, and then displaying the data with **printdata()**.

9. An industry seals lorry and taxi. Create a class **Automobile** that stores production date and price. From this class derive another two classes: **Lorry**, which adds weight capacity in kilogram and **Taxi**, which adds seat-capacity in number. Each of these classes should have member functions to get data and set data. Use user-defined constructors to initialize these objects.

10. Create a class called **cricketer** with member variables to represent name, age and no of matches played. From this class derive two classes: **Bowler** and **Batsman**. Bowler class has no_of_wickets as member variable and Batsman class has no_of_runs and centuries as member variables. Use appropriate member functions in all classes to read and display respective data.

11. Define a base class **Shape** having data member radius (int). Derive new classes called **Circle** and **Sphere** from this class. Write methods to compute the area of circle and sphere.

12. Create classes **Book** having data members name of author (string), price (float) and class **Stock** having data members number of books (int) and category (string). Create another class **Library** which derives from both the classes Book and Stock. All the classes should have functions having same name. Write program to test these classes.

Namespace

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name. The format of namespaces is:

namespace identifier
{
entities
}

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

Using scope resolution operator:
In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the "myNamespace" namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

Example program1

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

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.

Through keyword *using:*

The keyword using is used to introduce a name from a namespace into the current declarative region.

Example program2

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first
  cout << x << endl;
  cout << y << endl;
using namespace second
cout<<x<<endl;
cout<<y<<endl;
  return 0;
}
```

Operator Overloading

The concept of overloading can be applied to operators as well. Operator overloading is the mechanism of giving special meanings to an operator. It provides a flexible option for the operations of new definitions for most of the C++ operators. In other words, operator overloading refers to giving the normal C++ operators (such as +, *, <=, += etc) additional meanings when they are applied to user-defined data types. In general,

a = b + c; works only with basic types like 'int' and 'float', and attempting to apply it when a, b and c are objects of a user defined class will cause complaints from the compiler. But, using overloading, we can make this statement legal even when a, b and c are user defined types (objects).

There are two types of operator overloading

- Unary operator overloading and
- Binary operator overloading

Unary operator overloading

Unary operators are those operators that act on a single operand. ++, -- are unary operators. The following program overloads the ++ unary operator for the distance class to increment the data number by one.

```
class Distance
{
        int feet;
        float inch;
        public:
        Distance (int f, float i);
        void operator++(void);
        void display();
};
Distance :: Distance (int f, float i)
{
        feet = f ; inch = i;
}
void Distance :: display()
{
        cout<<"Distance in feet"<<feet<<endl;
        cout<<"Distance in inch"<<inch<<endl;
}
void Distance :: operator ++(void)
{
        feet++;
        inch++;
}
void main()
{
```

```
        Distance dist(10,10);
        ++dist;
        dist.display();
}
```

In the above example, the ++ unary operator has been overloaded in the function void Distance :: operator ++(void). In this overloaded function, data members feet and inch are increased by one. This function is called at the second line '++dist' in the main.

General syntax for defining operator overloading

```
return-type   classname :: operator  operator-to-overload (arg. list)
{
        //func body
}
```

The keyword '**operator**' is used to overload an operator. This declaration tells the compiler to call this member function whenever the ++ operator is encountered, provided the operands are of user-defined type.

//Another Example - overloading unary minus operator

```
class abc
{
        int x,y,z;
        public:
        void getdata(int a,int b,int c)
        {
                x = a;
                y = b;
                z = c;
        }
        void display()
        {
                cout<<x<<y<<z<<endl;
        }
        void operator -();
};
void abc::operator-()
{
        x = -x;
        y = -y;
        z = -z;
}

main()
{
        abc a;
        a.getdata(4,-5,6);
```

```
        a.display();
        -a;
        a.display();
        getch();
        return 0;
}
```

Operator Return Values

The operator++() function can return a value. If we use a statement like this

```
c1 = ++c2;
```

For this we have to define the ++ operator to have a return type object of a class in the operator++ function. That is the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1. the example given below illustrates this.

```
class Counter

{
        int count;
        public:
        Counter()
        {
                count = 0;
        }
        Counter(int c)
        {
                count = c;
        }

        Counter operator++()
        {
                return Counter(++count);
        }
        void put_count()
        {
                cout<<count<<endl;
        }
};
main()
{
        Counter c1,c2;
        c1.put_count();
        c2.put_count();
        c2 = ++c1;
```

```
        c1.put_count();
        c2.put_count();
        getch();
        return 0;
}
```

Postfix Notation

We have overloaded the ++ operator in prefix form. What about postfix, where the variable is incremented after its value is used in the expression?

c1++

to make both versions of the increment operator work, we define two overloaded ++ operators as follows

```
class Counter

{
        int count;
        public:
        Counter()
        {
                count = 0;
        }
        Counter(int c)
        {
                count = c;
        }

        Counter operator++()
        {
                return Counter(++count);
        }
        Counter operator++(int)
        {
                return Counter(count++);
         }
        void put_count()
        {
                cout<<count<<endl;
        }
};
main()
{
        Counter c1,c2,c3;
        c1.put_count();
```

```
        c2.put_count();
        c2 = ++c1;
        c3 = c2++;
        c1.put_count();
        c2.put_count();
        c3.put_count();
        getch();
        return 0;
}
```

Now there are two different declarators for overloading the ++ operator. The one, for prefix notation, is

Counter operator ++()

The one, for postfix notation, is

Counter operator ++(int)

The only difference is the **int** in the parentheses. This **int** is not really an argument, and it does not mean integer. It's simply a signal to the compiler to create the postfix version of the operator. The designers of C++ are fond of recycling existing operators and keywords to play multiple roles, and **int** is the one they chose to indicate postfix.

Overloading binary operators

Binary operators are those that work on two operands. Examples are +,-,*, /, % for arithmetic operations, +=,-=,*= and /= for assignment operations and >, <, <=,>=, == and! = for comparison operations.

Overloading a binary operator is similar to overloading unary operator except that a binary operator requires an additional parameter. The following code fragment overloads binary + operator. It adds two objects of type 'distance'.

```
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
```

```
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance operator + (Distance);
};
Distance Distance :: operator + (Distance d2)
{
        int m = meter + d2.meter;
        int cm = centimeter + d2.centimeter;
        if(cm >= 100)
        {
                cm -= 100;
                m++;
        }
        return Distance (m,cm);
}
main()
{
        Distance d1(4,50);
        Distance d2,d3,d4;
        d2.getDist();
        d3 = d1 + d2;  // Invokes operator+() function
        d4 = d3.operator+(d1); // usual function call syntax
        d3.show();
        d4.show();
        return 0;
}
```

NOTE Here the function operator +(Distance) does not need two arguments, since there are two objects to be added. The argument on the left side the operator (d1 here) is the object of which the operator is a member. The object on the right side of the operator (d2 here) must be furnished as an argument to the operator.

Overloading Binary Operators Using Friend Function

Friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be

explicitly passed to it, while a member function requires only one. The distance addition program discussed above can be modified using friend function as follows:

```
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        friend Distance operator + (Distance,Distance);
};
Distance operator + (Distance d1,Distance d2)
{
        int m = d1.meter + d2.meter;
        int cm = d1.centimeter + d2.centimeter;
        if(cm >= 100)
        {
                cm -= 100;
                m++;
        }
        return Distance (m,cm);
}
main()
{
        Distance d1(4,50);
        Distance d2,d3;
```

```
        d2.getDist();
        d3 = d1 + d2;
        d3.show();
        getch();
        return 0;
}
```

---------- Examples Programs----------

```
//Overloading == operator
class equal
{
        int feet;
        float inch;
        public:
        equal()
        {}
        equal(int f,float i)
        {
                feet = f;
                inch = i;
        }
        void display()
        {
                cout<<feet<<inch;
        }
        int operator ==(equal);
};
int equal::operator==(equal e)
{
    if(feet == e.feet && inch == e.inch)
    return 1;
    else
    return 0;
}

main()
{
        equal e1(6,4.4);
        equal e2(6,4.2);
        if(e1==e2)
        cout<<"Both are of same length";
        else
        cout<<"Not same";
        getch();
        return 0;
}
```

```
//Overloading == operator to compare two strings
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 20
class string
{
        private:
        char str[SZ];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char s[])
        {
                strcpy(str,s);
        }
        void getstring()
        {
                cout<<"\nEnter a string ";
                cin>>str;
        }
        int operator ==(string ss)
        {
                return(strcmp(str,ss.str) == 0)?1:0;
        }
        void display()
        {
                cout<<str<<endl;
        }
};

main()
{
        clrscr();
        string s1 = "Nepal";
        string s2 = "Kathmandu";
        string s3;
        s3.getstring();
        if(s3 == s1)
        cout<<"\nYou typed Nepal";
        else if(s3 == s2)
        cout<<"\nYou typed Kathmandu";
        else
        cout<<"\nNot of both";
```

```
	getch();
	return 0;
}
```

//Overloading + operator to concatenate two strings

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 20
class string
{
	private:
	char str[SZ];
	public:
	string()
	{
		strcpy(str," ");
	}
	string(char s[])
	{
		strcpy(str,s);
	}
	void display()
	{
		cout<<str<<endl;
	}
	string operator +(string ss)
	{
		string temp;
		if(strlen(str)+strlen(ss.str) < SZ)
		{
			strcpy(temp.str,str);
			strcat(temp.str,ss.str);
		}
		else
		{
			cout<<"\nstring overflow";
		}
		return temp;
	}
};

main()
{
	string s1 = "Shiva";
	string s2 = "Parbati";
```

```
        string s3;
        s1.display();
        s2.display();
        s3 = s1 + s2;
        s3.display();
        cout<<endl;
        getch();
        return 0;
}

//Overloading < operator
class line
{
        int feet;
        float inch;
        public:
        line()
        {}
        line(int f,float i)
        {
                feet = f;
                inch = i;
        }
        void display()
        {
                cout<<feet<<inch;
        }
        int operator<(line);
};
int line::operator<(line l)
{
        float l1 = feet + inch/12;
        float l2 = l.feet + l.inch/12;
        if(l1<l2)
                return 1;
        else
        return 0;
}

main()
{

        line l1(2,5.5);
        line l2(1,6.3);
        if(l1<l2)
                l1.display();
```

```
        else
                l2.display();
        getch();
        return 0;
}
```

Multiple Overloading

We have seen several different uses of + operator: - to add distance and to concatenate strings. We can put both these classes together in the same program, and C++ still knows how to interpret the + operator. It selects the correct function to carry out the addition based on the type of operand. Such an example is given below.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define SZ 40
class Distance
{
        int meter;
        int centimeter;
        public:
        Distance()
        {
                meter = 0;
                centimeter = 0;
        }
        Distance (int m, int cm)
        {
                meter = m;
                centimeter = cm;
        }
        void getDist()
        {
                cout<<"Enter meter";
                cin>>meter;
                cout<<"Enter centimeter";
                cin>>centimeter;
        }
        void show()
        {
                cout<<meter<<"\t"<<centimeter;
        }
        Distance operator + (Distance);
};
Distance Distance :: operator + (Distance d2)
{
```

```
        int m = meter + d2.meter;
        int cm = centimeter + d2.centimeter;
        if(cm >= 100)
        {
                cm -= 100;
                m++;
        }
        return Distance (m,cm);
}
class string
{
        private:
        char str[SZ];
        public:
        string()
        {
                strcpy(str," ");
        }
        string(char s[])
        {
                strcpy(str,s);
        }
        void display()
        {
                cout<<str<<endl;
        }
        string operator +(string ss)
        {
                string temp;
                if(strlen(str)+strlen(ss.str) < SZ)
                {
                        strcpy(temp.str,str);
                        strcat(temp.str,ss.str);
                }
                else
                {
                        cout<<"\nstring overflow";
                }
                return temp;
        }
};

main()
{

        Distance d1(4,50);
```

```
        Distance d2,d3;
        d2.getDist();
        d3 = d1 + d2;
        d3.show();
        string s1 = "Shiva";
        string s2 = "Parbati";
        string s3;
        s1.display();
        s2.display();
        s3 = s1 + s2;
        s3.display();
        getch();
        return 0;
}
```

General rules for overloading operators

- There are some restrictions and limitations to be kept in mind while overloading operators. They are as follows:
- Only the existing operators can be overloaded. New operators can not be created.
- The overloaded operators must have at least one user-defined operand.
- It is not recommended to change the basic meaning of an operator. That is, the plus (+) operator should not be redefined to subtract one value from another.
- Overloaded operators follow syntax rules of the original operators. That can not be overridden.
- Friend functions can not be used to overload certain operators like =, ( ), [ ] and ->.
- Unary operators, overloaded by means of a member functions take no explicit arguments and return no explicit values. But those overloaded by a friend functions take one reference argument.
- Binary operator overloaded through a member function take one explicit argument and those that are overloaded through a friend function take two arguments.
- Binary operators such as +, -, *, and / must explicitly return a value.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

Operator Overloading Restriction

- Operator functions can not have default arguments.
- The operators that can not be overloaded are **.** , **::**, **.***, ?:, sizeof.
- One can not alter the precedence of operators
- One can not change the number of operands that an operator takes.

Type Conversion (Data Conversion)

We use the assignment operator (=) to assign value of one variable to another. For example

x = y;

Where x and y are integer variables. We have also noticed that = assigns the value of one user defined object to another, provided that they are of the same type. For example,

d2 = d1;

Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler does not need any special instructions to use = for the assignment of user-defined objects such as distance objects.

The assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign.

But if the variables on different sides of the = are of different types, then the type of variable on the right side of = needs to be converted to the type of left side variable before the assignment takes place.

Type conversion is the conversion of one data type to another data type.

Conversion Between Basic Types

Consider the statement,

intvar = floatvar;

where intvar is of type int and floatvar is of type float. Here the compiler will call a special routine to convert the value of floatvar, which is expressed in floating point format, to an integer format so that it can be assigned to intvar. There are many such conversions: from **float** to **double**, **char** to **float** and so on. Each such conversion has its own routine, built into the compiler and called up when the data types on different sides of the = sign so dictate. Such conversions are **implicit conversion**.

Sometimes we want to force the compiler to convert one type to another. For example,

```
int total = 400;
float avg;
avg = float(total) / 5;  // converts value of total to float before division takes place.
```

Conversion Between Objects and Basic Types

When we want to convert between user-defined data types and basic types, we can not rely on built-in conversion routines, since the compiler does not know anything about user-defined types besides what we tell it. Instead, we must write these routines ourselves.

From Basic to Class type Conversion

To go from a basic type to a user defined type, we use constructor. These are sometimes called ***conversion constructors.*** Example-

```
class time
{
        int hrs;
        int min;
        public:
        time()
        {}
        time(int t)
        {
                hrs = t/60;
                min =  t%60;
        }
        void display()
        {
                cout<<"Hours = "<<hrs<<"Minutes = "<<min;
        }
};

void main()
{
  time  t1 = 95; // uses one-argument constructor to convert integer to time
  t1.display();
  getch();

}
```

From User-Defined to Basic Type

When class type data is converted into basic type data, it is called class to basic type conversion. The constructor functions do not support this operation. This type of conversion takes place in casting operator. The casting operator function is also called as conversion function. The syntax of casting operator function is

```
operator typename()
{
        ……..
        //function body
```

```
        ……..
}
```

This function converts a class type data to typename. For example- the operator double() converts class object to type double. The operator int() converts a class type object to type int and so on.

The casting operator function should satisfy the following conditions

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

```
class Stock
{
        int items;
        float price;
        public:
        Stock(int a,float p)
        {
                items = a;
                price = p;
        }
        void putdata()
        {
                cout<<"Items: "<<items<<"\n";
                cout<<"Price: "<<price<<"\n";
        }
        operator float()
        {
                return (items*price);
        }

};
main()
{
        Stock s(45,2.5);
        float total_value;
        total_value = s;
        cout<<"\nData of s ";
        s.putdata();
        cout<<"Total float value = "<<total_value;
        getch();
        return 0;
}
```

Another example

```
class DistConv
{
        private:
        int kilometers;
        double meters;
        static double kilometersPerMile;
        public:
   // This function converts a built-in type (i.e. miles) to the
   // user-defined type (i.e. DistConv)
        DistConv(double mile)  // Constructor with one argument
        {
                double km = kilometersPerMile * mile ;    // converts miles to
                                //kilometers

                kilometers = int(km);           // converts float km to
                                //int and assigns to kilometer

                meters = (km - kilometers) * 1000 ;    // converts to meters
        }
        DistConv(int k, float m)   // constructor with two arguments
        {
                kilometers = k ;
                meters = m ;
        }
   // ********Conversion Function************
   operator double()              // converts user-defined type i.e.
                           // DistConv to a basic-type
   {                         // (double) i.e. meters
      double K = meters/1000 ;       // Converts the meters to
                              // kilometers
      K += double(kilometers) ;       // Adds the kilometers
      return K / kilometersPerMile ;    // Converts to miles
   }

   void display(void)
   {
      cout << kilometers << " kilometers and " << meters << " meters" ;
   }
}; // End of the Class Definition

double DistConv::kilometersPerMile = 1.609344;

int main(void)
{
```

```
   DistConv d1 = 5.0 ;              // Uses the constructor with one argument


   DistConv d2( 2, 25.5 );          // Uses the constructor with two arguments
   double ml = double(d2) ;         // This form uses the conversion function
                                    // and converts DistConv to miles
   cout << "2.255 kilometers = " << ml << " miles\n" ;
   ml = d1 ;                        // This form also uses conversion function
                                    // and converts DistConv to miles

   d1.display();
   cout << " = " << ml << " miles\n" ;
   getch();
}

/*Output
2.255 kilometers = 1.25859 miles
8 kilometers and 46.72 meters = 5 miles*/
```

From One Class to Another Class Type

When a data of one class type is converted into data of another class type, it is called conversion of one class to another class type. For example-

objx = objy;

here objx is an object of class X and objy is object of class Y. The class Y type data is converted to the class X type data and converted value is assigned to the objx. Since the conversion takes place from class Y to class X, Y is known as source class and X is known as destination class. This type of conversion is carried out by either constructor or a conversion function. Then how do we decide which form to use? It depends upon where we want the type conversion function to be located in the source class or in the destination class.

We know that the casting operator function

operator typename()

Converts the class object of which it is a member to typename. The typename may be a built-in type or user-defined type one. In the case of conversion between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. This implies that the argument belongs to the source class and is passed to the destination class for

conversion. This makes it necessary that the conversion constructor be placed in the destination class.

```
class Kilometers
{
        private:
        double kilometers;
        public:
         Kilometers(double kms)
        {
                kilometers = kms;
        }
        void display()
        {
                cout << kilometers << " kilometeres";
        }
        double getValue()
        {
                return kilometers;
        }
};

class Miles
{
        private:
        double miles;
        public:
        Miles(double mls)
        {
                miles = mls;
        }
        void display()
        {
                cout << miles << " miles";
        }
        operator Kilometers()
        {
                return Kilometers(miles*1.609344);
        }
        Miles(Kilometers km)
        {
                miles = km.getValue()/1.609344;
        }
};
```

```
int main(void)
{
   /*
    * Converting using the conversion function
    */
   Miles m1 = 100;
   Kilometers k1 = m1;

   m1.display();
   cout << " = ";
   k1.display();
   cout << endl;

   /*
    * Converting using the constructor
    */
   Kilometers k2 = 100;
   Miles m2 = k2;    // same as:  Miles m2 = Miles(k2);
   k2.display();
   cout << " = ";
   m2.display();
   cout << endl;
   getch();
   return 0;
}
/*Output
100 miles = 160.934 kilometeres
100 kilometeres = 62.1371 miles
*/


class invent1
{
        int code;
        int items;
        float price;
        public:
        invent1(int a,int b,float c)
        {
                code = a;
                items = b;
                price = c;
        }
        void putdata()
        {
                cout<<"Code: "<<code<<"\n";
```

```
		cout<<"Items: "<<items<<"\n";
		cout<<"Value: "<<price<<"\n";
	}
	int getcode()
	{
		return code;
	}
	int getitems()
	{
		return items;
	}
	int getprice()
	{
		return price;
	}
	operator float()
	{
		return (items*price);
	}

};

class invent2
{
	int code;
	float value;
	public:
	invent2()
	{
		code = 0;
		value = 0;
	}
	invent2(int x,float y)
	{
		code = x;
		value = y;
	}
	void putdata()
	{
		cout<<"Code: "<<code<<"\n";
		cout<<"Value: "<<value<<"\n";
	}
	invent2(invent1 p)
	{
		code = p.getcode();
		value = p.getitems() * p.getprice();
```

```
        }
};

main()
{
        clrscr();
        invent1 s1(100,5,140.0);
        invent2 d1;
        float total_value;
        /*invent to float*/
        total_value = s1;
        /*invent1 to invent2*/
        d1 = s1;
        cout<<"Product details - invent1 type"<<"\n";
        s1.putdata();
        cout<<"Stock value"<<"\n";
        cout<<"value = "<<total_value<<"\n";
        cout<<"Product details-invent2 type"<<"\n";
        d1.putdata();
        getch();
        return 0;
}
```

Lab Sheet-2

8. Write a program to overload += operator. (You can overload this operator using Distance class as d1 += d2)

9. Write a program to overload = operator (It is already overloaded in C++). Use Distance class to test the program.

10. Write a program to overload ++ operator using friend function.

11. Create a class called **Length** that has data members meter and centimeter. Overload + operator to add two objects of class Length. (For example L3 = L1 + L2). Also facilitate the operations like L4 = L1 + 5 and L5 = 5 + L4 where L1, L2, L3, L4 and L5 are objects of class Length. Use constructors and member functions to initialize and display values.

12. Write a conversion routine in c++ that can convert user-defined data distance to basic data float. Assume that the class distance contains two data members (feet (integer type) and inch (floating point type). NOTE 1-meter = 3.33 feet and 1 feet = 12 inches)

13. Define a class to hold rectangular co-ordinates, i.e. x and y co-ordinates. Let P1 and P2 be the objects of this class where P1 is initialized to (20, 30). Facilitate the operation P2 = P1++ in such a way that the value in P2 is (21, 31) afterward.

14. Write a program to overload + operator to concatenate two strings.

Virtual Function and Run Time Polymorphism

Pointers

Pointers have a reputation for being hard to understand. One important use for pointers is in the dynamic allocation of memory, carried out in C++ with the keyword new and its partner delete

Addresses (Pointer Constants)

Every byte in the computer's memory has an address. Addresses are numbers, just as they are for houses on a street.
The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575; for 16 MB of memory, it is 16,777,215.

The Address of Operator &

You can find out the address occupied by a variable by using the address of operator &.

New and Delete Operator

Pointer provides the necessary support for C++ powerful dynamic memory allocation system. Dynamic allocation is the means by which a program can obtain memory while it is running.

For eg- int arr[100];

reserves memory for 100 integers. Arrays are a useful approach to data storage, but they have a serious drawback. We must know at the time we write the program how big the array will be and it is not always possible to predict what the size of the array will be. It would be desirable to start the program and then allocates memory as the need arises. This capability is provided by the new operator. This versatile operator obtains memory from the operating system and returns a pointer to the starting point. The syntax for the new operator is
<variable> = new <type>;
where <variable> = pointer variable
<type> = char, int, float and so on

type of variable mentioned on the left hand side and the type mentioned on the right hand side should match. For eg-

```
char * cptr;
cptr = new char;
int *iptr;
iptr = new int;
```

the syntax of new operator can also be modified to allocate memory of varying requirements. For eg-

```
char *cptr;
cptr = new char[10];
```

allocates 10 bytes of memory and assigns the starting address to cptr.

Delete Operator

If our programs reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system.

Deleting the memory does not delete the pointer that points to it and does not change the address value in the pointer. However, this address is no longer valid, the memory it points to may be changed to something entirely different but we do not use pointers memory that has been deleted.

Syntax:

delete <variable>;

Where <variable> = pointer variable

Eg- student *ps;

ps = new student;

delete ps;

If we are deleting an array, we use bracket following delete. For eg-

delete[] cptr;

```
//Example program that makes use of new and delete operator
#include<iostream.h>
#include<conio.h>
class test
{
  int x;
  public:
  void input()
  {
        cin>>x;
  }
  void display()
  {
        cout<<x;
  }
};
main()
{
  test *t = new test;
  t->input();
  t->display();
  delete t;
  getch();
  return 0;
}
```

Polymorphism

In the programming sphere, polymorphism is broadly divided into two parts- the first part being static polymorphism- exhibited by overloaded functions and the second being dynamic polymorphism exhibited by late binding.

Static Polymorphism

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions on the basis of number, type, and sequence of their arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to the calls at compile time. This form of association is called early binding. The term early binding stems from the fact that when the program is executed, the calls are already bound to the appropriate functions. The resolution is on the basis of number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration.

void add(int, int);
void add(float, float);

Now, if the function add() is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term late binding refers to the resolution of the function to their associated methods at run time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context. The compiler is unable to bind a call to a method since resolution depends on the context of the call.

Static binding is considered to be more efficient and dynamic binding more flexible.

Virtual Functions

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. A function is made virtual by placing the keyword **virtual** before its normal declaration.

Normal Member Functions Accessed with Pointers

What will happen when a base class and the derived classes all have functions with the same name, and we access these functions using pointers but without using virtual functions? Here is the program for this.

```
#include<iostream.h>
#include<conio.h>
class Base
{
```

```
        public:
        void show()
        {
                cout<<"Base\n";
        }
};
class Derv1 : public Base
{
        public:
        void show()
        {
                cout<<"Derv1\n";
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
                cout<<"Derv2\n";
        }
};
int main()
{
        Derv1 d1;
        Derv2 d2;
        Base *ptr; // pointer to base class
        ptr = &d1;
        ptr->show();
        ptr = &d2;
        ptr->show();
        getch();
        return 0;
}
```

Output of the program
Base
Base
The function in the base class is always executed. The compiler ignores the contents of the pointer ptr and chooses the member function that matches the type of the pointer.

Virtual Member Functions Accessed with Pointers
Let's make a single change in the above program. We will place the keyword **virtual** in front of the declarator for the **show()** function in the base class.

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

```
class Base
{
	public:
	virtual void show()
	{
		cout<<"Base\n";
	}
};
class Derv1 : public Base
{
	public:
	void show()
	{
		cout<<"Derv1\n";
	}
};
class Derv2 : public Base
{
	public:
	void show()
	{
		cout<<"Derv2\n";
	}
};
int main()
{
	Derv1 d1;
	Derv2 d2;
	Base *ptr;
	ptr = &d1;
	ptr->show();
	ptr = &d2;
	ptr->show();
	getch();
	return 0;
}
```

Output of the program

```
Derv1
Derv2
```

Now, as you can see, the member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of Derv1 to that of Derv2, and the particular instance of show() that is executed also changes. So the same function call,

```
ptr->show();
```

executes different functions, depending on the contents of ptr. The compiler selects the function according to the contents of the pointer ptr, not on the type of the pointer. Here, the compiler does not know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class. Which version of show() does the compiler call? In fact the compiler does not know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointing to by ptr, the appropriate version of show() will be called, exhibiting late binding.

Abstract Classes and Pure Virtual Functions

An abstract class is one that is not used to create objects. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. A class is made an abstract by placing at least one pure virtual function in the class. A pure virtual function is one with expression = 0 added to the declaration. i.e. a pure virtual function can be declared by equating it to zero.

For example, to make show() function virtual we write

virtual void show ()= 0; // pure virtual function

the equal sign here has nothing to do with assignments, the value 0 is not assigned to anything. The =0 syntax is simply how we will tell the compiler that a function will be pure.

Once we have placed a pure virtual function in the base class, then we must override it in all the derived classes from which we want to instantiate objects. If a class does not override the pure virtual function, then it becomes an abstract class itself, and we can not instantiate objects from it. For consistency, we make all the virtual functions in the base class pure.

We can not create objects of the abstract class. However, we can create pointers to an abstract class. This allows an abstract class to be used as a base class, pointers to which can be used to select the proper virtual function.

```
//Program: Pure virtual function
#include<iostream.h>
#include<conio.h>
class Base
{
        public:
        virtual void show() = 0;
};
class Derv1 : public Base
{
        public:
        void show()
        {
                cout<<"Derv1\n";
```

```
        }
};
class Derv2 : public Base
{
        public:
        void show()
        {
                cout<<"Derv2\n";
        }
};
int main()
{
        //Base b; // can't make object of abstract class
        Base *arr[2];
        Derv1 d1;
        Derv2 d2;
        arr[0] = &d1;
        arr[1] = &d2;
        arr[0]->show();
        arr[1]->show();
        getch();
        return 0;
}
```

Output of the program

```
Derv1
Derv2

//Example program: pure virtual function
#include<iostream.h>
#include<conio.h>
class person
{
        protected:
        char name[20];
        public:
        void getName()
        {
                cin>>name;
        }
        void putName()
        {
                cout<<endl<<name;
        }
        virtual void getData() = 0;
        virtual int isOutstanding() = 0;
};
```

```
class student : public person
{
	private:
	float gpa;
	public:
	void getData()
	{
		person::getName();
		cout<<"\nEnter students gpa";
		cin>>gpa;
	}
	int isOutstanding()
	{
		return (gpa > 3.5) ? 1 : 0;
	}
};
class professor : public person
{
	private:
	int numpubs;
	public:
	void getData()
	{
		person :: getName();
		cout<<"\nEnter number of publications";
		cin>>numpubs;
	}
	int isOutstanding()
	{
		return (numpubs > 100)?1 : 0;
	}
};
main()
{
	person *persptr[100];
	int n = 0;
	char choice;
	do
	{
		cout<<"\nEnter student or professor(s/p)";
		cin>>choice;
		if(choice == 's')
			persptr[n] = new student;
		else
			persptr[n] = new professor;
		persptr[n++]->getData();
```

```
                cout<<"\nAnother record? ";
                cin>>choice;
        }while(choice == 'y');

        for(int j = 0;j < n; j++)
        {
                persptr[j]->putName();
                if(persptr[j]->isOutstanding())
                cout<<"\nThis person is outstanding";
        }
        getch();
        return 0;
}
```

Friend class

The member functions of a class can all be made friends at the same time when we make the entire class a friend. Consider the following program.

```
#include<iostream.h>
#include<conio.h>
class alpha
{
        int data;
        public:
        alpha()
        {
                data = 10;
        }
        friend class beta;  // beta is friend class
};
class beta
{
        public:
        void func1(alpha a)
        {
                cout<<a.data;
        }
        void func2(alpha a)
        {
                cout<<a.data;
        }

};
main()
{
        alpha a;
        beta b;
```

```
        b.func1(a);
        b.func2(a);
        getch();
        return 0;
}
```

In class **alpha** the entire class **beta** is defined as a friend. Now all the member functions of **beta** can access the private data of **alpha**.

The this Pointer

The member functions of every object have access to a sort of magic pointer named **this**, which points to the object itself. Consider the program

```
#include<iostream.h>
#include<conio.h>
class MyClass
{
        public:
        void TestThisPointer()
        {
                cout<<"\nMy object's address is"<<this;
        }
};
main()
{
        MyClass m1,m2;
        m1.TestThisPointer();
        m2.TestThisPointer();
        getch();
        return 0;
}
```

Accessing Member Data with this

When we call a member function, it comes into existence with the value of **this** set to the address of the object for which it was called. The **this** pointer can be treated like any other pointer to an object, and can thus be used to access the data in the object it points to.

```
class MyClass
{
        int x;
        public:
        void test()
        {
                this->x = 10;
                cout<<this->x;
```

```
        }
};
main()
{
        MyClass m;
        m.test();
        getch();
        return 0;
}
```

This program simply prints the value 10. The test() member function accesses the variable **x** as

this->x

This is exactly the same as referring to **x** directly.

Using this for Returning Values

A more practical use for **this** is in returning values from member functions and overloaded operator.

```
class alpha
{
        int data;
        public:
        alpha()
        {}
        alpha(int x)
        {
                data = x;
        }
        void display()
        {
                cout<<data<<endl;
        }
        alpha& operator = (alpha &a)
        {
                data = a.data;
                return *this;
        }
};
main()
{
        alpha a1(50);
        alpha a2;
        a2 = a1; // calls overloaded operator =
        a1.display();
```

```
	a2.display();
	getch();
	return 0;
}

//using this pointer for returning value from member function
class Distance
{
	int meter;
	int centimeter;
	public:
	Distance()
	{
		meter = 0;
		centimeter = 0;
	}
	Distance (int m, int cm)
	{
		meter = m;
		centimeter = cm;
	}
	void getDist()
	{
		cout<<"Enter meter";
		cin>>meter;
		cout<<"Enter centimeter";
		cin>>centimeter;
	}
	void show()
	{
		cout<<meter<<"\t"<<centimeter;
	}
	Distance compare(Distance);
};
Distance Distance :: compare(Distance d2)
{
	float dist1 = meter + (float)centimeter/100;
	float dist2 = d2.meter + (float)d2.centimeter/100;
	if(dist1 < dist2)
		return *this;
	else
		return d2;
}
main()
{
	Distance d1(4,50);
```

```
        Distance d2,d3,d4;
        d2.getDist();
        d4 = d1.compare(d2);
        cout<<"Small lenght is: ";
        d4.show();
        getch();
        return 0;
}
```

Templates

(Meaning to word: A document or file or entity having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used)

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.

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

A template is a way to specify generic code, with a placeholder for the type. Note that the type is the only "parameter" of a template, but a very powerful one, since anything from a function to a class (or a routine) can be specified in "general" terms without concerning yourself about the specific type.

Templates offer several advantages:

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

Templates and Macros

In many ways, templates work like preprocessor macros, replacing the templated variable with the given type. However, there are many differences between a macro like this:

```
#define min(i, j) (((i) < (j)) ? (i) : (j))
```

and a template:

```
template<class T> T min (T i, T j) { return ((i < j) ? i : j) }
```

Here are some problems with the macro:

- There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.
- The i and j parameters are evaluated twice. For example, if either parameter has a post incremented variable, the increment is performed two times.
- Because macros are expanded by the preprocessor, compiler error messages will refer to the expanded macro, rather than the macro definition itself. Also, the macro will show up in expanded form during debugging.

Templates in c++ comes in two variations

a) function templates
b) class templates

Class template

The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

The general form of a class template is

Template <class T>
Class class_name
{
//class member with type T whenever appropriate
};

Example

```
#include <iostream.h>

template<class T>
class vec {

        T x;
        T y;

public:
        vec(T f1, T f2)
        {
          x=f1;
          y=f2;
   }

        vec()
         {  }

 vec operator+(const vec& v1)
 {
   vec result;
   result.x = v1.x+this->x;
   result.y = v1.y+this->y;
   return result;
 }
};

int main() {
 vec<int> v1(3,6);
 vec<int> v2(2,-2);
```

```
 vec<int> v3=v1+v2;


 vec<float> v4(3.9,6.7);
 vec<float> v5(2.0,-2.2);
 vec<float> v6=v4+v5;

}
```

Advantages of C++ Class Templates:

- One C++ Class Template can handle different types of parameters.
- 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.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

Function template

To perform identical operations for each type of data compactly and conveniently, we use function templates. we can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

Syntax:

The general form of a function template is

```
Template<class T>
returnType function_name (argument of type T)
{
//body of function with type T whenever appropriate
}
```

Using Template Functions: example

Using function templates is very easy: just use them like regular functions. When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max(int, int). Similarly the compiler generates definitions for max(char, char) and max(float, float) in this case.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements

template <class T>          //function template
```

```
T max(T a, T b)
{
        return a > b ? a : b ;
}

int main()
{

        cout << "max(10, 15) = " << max(10, 15) << endl ;
        cout << "max('k', 's') = " << max('k', 's') << endl ;
        cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
        return 0;
}
```

Output:

max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2

Explanation:

The template keyword signals the compiler that I'm about to define a function template. The keyword class, within the angle brackets, might just as well be called type. As you've seen, you can define your own data types using classes, so there's really no distinction between types and classes. The variable following the keyword class (T in this example) is called the *template argument*.

Standard Template Library(STL)