Sonali Sagar

C++

Language Basics:

Generally languages are classified as follows:

1. Procedure oriented programming languages.
   e.g. C, Pascal etc.
2. Object Oriented Programming languages.
   e.g. Simula, Smalltalk, C++, Java, C# etc.
3. Object Based Programming Languages.
   e.g. ada, modula-2, visual Basic
4. Logic Oriented Programming Languages.
5. Rule Oriented Programming Languages.
6. Constraint Oriented Programming Languages.

Advantages of object oriented programming:

1. Code Reusability

Objects created for object oriented programs can easily be reused in other programs.

2. Lower Cost Of Development

More effort is put into the object oriented analysis and design which lowers overall cost of development.

3. Improved Software Maintainability

Since design is modular, part of the system can be updated in case of issues without a need to make large scale changes.

4. Higher Quality Software

Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software which tends to result in higher quality software.

Parts/Elements/Pillars of OOP model:

There are four major and three minor elements of object oriented programming model.

Major means a language without any one of these elements is not considered as object oriented.

Following are the major pillars of OOPs:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

Minor means, each of these elements is useful but not essential.

Following Are the minor pillars of OOP:

1. Typing
2. Concurrency
3. Persistence

Brief history of C++:

The C++ programming language has a history going back to 1979, when Bjarne Stroustrup was doing work for his Ph.D. thesis. One of the language Stroustrup had the opportunity to work with was a language called Simula, which as the name implies is a language primarily designed for simulations. The Simula 67 language is regarded as the first language to support the object-oriented programming paradigm. Stroustrup found that this paradigm was very useful for software development, however the Simula language was far too slow for practical use.

Shortly thereafter, he began work on "C with Classes", which as the name implies was meant to be a superset of the C language. His goal was to add object-oriented programming into the C language, which was and still is a language well-respected for its portability without sacrificing speed or low-level functionality.

The first C with Classes compiler was called Cfront. It was a program designed to translate C with Classes code to ordinary C. Cfront would later be abandoned in 1993 after it became difficult to integrate new features into it, namely C++ exceptions. Nonetheless, Cfront made a huge impact on the implementations of future compilers and on the Unix operating system.

In 1983, the name of the language was changed from C with Classes to C++.

Data types:

Data type in any language describes three things:

1.How much memory is required to store the data.

2.Which kind of data that memory can hold.

3.Which operations we can perform on that data.

Basically there are only two types of data types but for our convenience we will classify it into three types

-Fundamental data types

1.void	unknown size	
2.bool	1 byte	[can contain either true or false value]

3.char	1 byte	[ASCII Compatible]
4.wchar_t	2 bytes	[Unicode Compatible]
5.int	4 bytes	
6.float	4 bytes	
7.double	8 bytes	

-Derived Datatypes

1.array
2.function
3.pointer
4.reference

-User Defined Data Types

1.Structure
2.Union
3.Class

Access Specifers in C++

There are three acess specifiers in c++

1.private:

If we declare elements of the class as private then we can access these members inside same class in which it is declared. We cannot access private members inside nonmember function. Generally data members should be declared as private.

2.protected:

If we declare elements of the class as protected then we can access these members inside same class in which it is declared as well as inside derived class. We cannot access protected members outside the class.

3.public:

If we declare elements of the class as public then we can access these members inside same class in which it is declared, inside derived class as well as inside non member function. Generally member function should be declared as public.

Structure in C++

-structure is collection of related elements which get continuous memory space.

-in case of instantiation use of struct keyword is optional.

-we can write variable as well as function inside structure in c++.

-by default members of the structure are treated as a public.

-to create light weight object we should use structure.

Data member:

-Variable declared inside class is called data member.

-Data member is also called field, property or attribute.

-If we create object then it gets single copy of the data member i.e. data member gets space inside object.

Member function:

-Function implemented inside a class is called member function.

-Member function is also called method, operation,behavior or message.

-Member function do not get space inside object rather all the objects of same type share single copy of member function.

Class:

-syntactically class is collection of data member and member function.

-In general 'Class is collection of such objects which represents common structure and common behavior'.

-Since object is always created by looking toward the, class is considered as a template/model/blueprint for an object.

-class is logical entity.

-class is a user defined data type. It is also called abstract data type.

Object:

-Syntactically, object is variable of class.

-An entity which is having physical existence is called object.

-In object oriented way, we can define object as a instance of class.

-At the time of creation of object use of class keyword is optional.

```
class Point pointInstance; //valid
Point  pointInstance;              //valid
```

Some important terminology

1. **Instantiation** - Process of creating object from a class is called instantiation.
2. **Concrete class** - If class allows us to create an object of a class then it is called concrete class.
3. **Concrete method** - Member function of a class which is having a body is called concrete method.

4. **Abstract class** - If class do not allows us to create an object of a class then it is called abstract class.

5. **Abstract method** - Member function of a class which do not contain body is called concrete method.

6. **Message Passing -** Process of calling member function on object is called message passing.

consider the following example

```
Point pt1;
pt1.Point::printRecord( ); //valid : Message passing
pt1.printRecord( );                //valid : Message passing
```

Characteristics of an Object:

1.State - The state of an object encompasses all of the properties of the object plus the current values of each of these properties. Values stored inside object is called state.

2.Behavior - Behavior is how an object acts and reacts, in terms of its state changes and message passing. Member functions of the class represent behavior of the object.

3.Identity - Identity is that property of an object which distinguishes it from all other objects.

An object is an entity that has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable.

Empty class and object:

A class which do not conain any member is called empty class.consider the following code.

```
class Empty
{
        //Empty implementation
};
```

According to Bjame stroustrup, to differentiate object from a class, object must get space inside memory and it should be non zero. But compilers do optimization because there is no use of such memory. So *size of object of empty class is one byte.*

this pointer:

- this is a keyword in C++.
- It is implicit pointer available inside every non static member function of the class, which is used to store address of current object.
- this is constant pointer and its general type is as follows

  ClassName* const this
- this represent address of current object while (*this)represents current object itself.
- Data member and member function can communicate with each other with the help of this pointer so this is considered as connection / link between data member and member function.
- If any member function accepts parameters then this pointer is always pass as a first parameter implicitly.
- We cannot declare this as a function parameter explicitly.
- Use of this keyword is optional but we should use it.
- If name of data member and function parameter is same then it is mandatory to use this keyword before data member.
- Following function do not get this pointer:
  1. Global function
  2. Static member function
  3. Friend function.

Array of Objects:

```
Point arr[ 3 ];         //Here parameterless ctor will call three times
for( int index = 0; index < 3; ++ index )
        arr[ index ].printRecord();

Point arr[ 3 ] = { Point(10,20),Point(30,40),Point(50,60)};
                //Here parameterized constructor will call three times
for( int index = 0; index < 3; ++ index )
        arr[ index ].printRecord();
```

Namespace :

Namespace is logical term, which is used to control scope of global members. In other words to avoid name ambiguity or name clashing or name collision we should use namespace.

```
int num1 = 10;
int num1 = 20;      //error: 'num1' : redefinition; multiple initialization
```

```
int main( void )
{
	printf("Num1	:	%d\n",num1);
	return 0;
}
```

As shown in above code we cannot give same name to multiple members in same scope. Problem may occur, if name of elements refereed from different library and name of element declared in same program is same. To avoid this problem we should implement global members inside namespace. we can write above program as follows

```
namespace NA
{
	int num1 = 10;
}
namespace NB
{
	int num1 = 20;	//Now no error
}
int main( void )
{
	printf("Num1	:	%d\n",NA::num1); //Num1	:	10
	printf("Num1	:	%d\n",NB::num1); //Num1	:	20
	return 0;
}
```

As shown in above code, we should use namespace name and scope resolution operator to access elements of namespace.

We can implement namespace inside scope of another namespace. It is called nested namespace. First we will see nested namespace with different element names.

```
int num1 = 10;
namespace NA
{
	int num2 = 20;
	namespace NB
	{
		int num3 = 30;
```

```
	}
}
int main(void)
{
	printf("Num3	:	%d\n", NA::NB::num3);		//Num3	:	30
	printf("Num2	:	%d\n", NA::num2);			//Num2	:	20
	printf("Num1	:	%d\n", ::num1);			//Num1	:	10
	return 0;
}
```

Generally we use the statement that to access global variable or function we should use :: operator.

In namespace topic we have discussed that to access elements of namespace we should :: operator. Consider above code. Since num3 is member of NB namespace & NB namespace is member of NA namespace we have written NA::NB::num3. num2 is member of NA namespace, we have written NA::num2.Even though num1 is not a member of any namespace we can use :: operator with it. Meaning of this statement is that, if we declare or implement global members without namespace it is considered as member of unnamed or global namespace.

If we want to use any member of namespace frequently then instead of specifying namespace name every time we can use using directive. The using directive allows the names in a namespace to be used without the namespace name as an explicit qualifier. consider the following code.

```
int num1 = 10;
namespace NA
{
	int num2 = 20;
	namespace NB
	{
		int num3 = 30;
	}
}
int main(void)
{
	printf("Num1	:	%d\n", ::num1);			//Num1	:	10
	using namespace NA;
```

```
        printf("Num2    :    %d\n", num2);          //Num2    :    20
        printf("Num3    :    %d\n", NB::num3);      //Num3    :    30
        using namespace NB;
        printf("Num3    :    %d\n", num3);          //Num3    :    30
        return 0;
}
```

We can write "using namespace namespace_name" statement any number of times.

We can give same name to the outer namespace as well as inner namespace. In this case we cannot use 'using' directive to access namespace members.

```
namespace NA
{
        int num1 = 10;
        namespace NA
        {
                int num2 = 20;
        }
}



int main(void)
{
        printf("Num1    :    %d\n", NA::num1);          //Num1    :    10
        printf("Num2    :    %d\n", NA::NA::num2);      //Num2    :    20
        return 0;
}
```

We can give same namespace name to other namespace inside same scope.

```
namespace NA
{
        int num1 = 10;
}
namespace NA
{
        int num2 = 20;
}
int main(void)
```

```
{
        printf("Num1    :       %d\n", NA::num1);       //Num1    :       10
        printf("Num2    :       %d\n", NA::num2);       //Num2    :       20

        using namespace NA;
        printf("Num1    :       %d\n", num1);           //Num1    :       10
        printf("Num2    :       %d\n", num2);           //Num2    :       20
        return 0;
}
```

we should use using statement carefully otherwise we may get unexpected results. consider following code snippet.

```
namespace NA
{
        int num1 = 10;
}
int main(void)
{
        int num1 = 20;
        using namespace NA;
        printf("Num1    :       %d\n", num1);           //Num1    :       20
        printf("Num1    :       %d\n", num1);           //Num1    :       20

        printf("Num1    :       %d\n", NA::num1);       //Num1    :       20
        printf("Num1    :       %d\n", num1);           //Num1    :       20
        return 0;
}
```

If name of members of namespace is same and if we try to access it with the help of using directive then compiler produces ambiguity error. In this case we should use fully qualified name. consider following code.

```
namespace NA
{
        int num1 = 10;
        namespace NB
        {
                int num1 = 20;
                int num2 = 30;
```

```
        }
}
int main(void)
{
        using namespace NA;
        printf("Num1    :    %d\n",num1);          //Num1    :    10
        using namespace NB;
        //printf("Num1    :    %d\n", num1);       //ambiguity error
        printf("Num1    :    %d\n", NB::num1);     //Num1    :    20
        printf("Num1    :    %d\n", num2);         //Num2    :    30
        return 0;
}
```

We can declare using directive at local scope as well as file scope.

In following code snippet we have declared "using namespace namespace_name" statement at local scope.

```
namespace NA
{
        int num1 = 10;
}
void show_record(void)
{
        using namespace NA;
        printf("Num1    :    %d\n", num1);
}
void display_record(void)
{
        using namespace NA;
        printf("Num1    :    %d\n", num1);
}
int main(void)
{
        ::show_record();
        ::display_record();
```

```
        return 0;
}
```

If we want to access namespace elements at multiple location then we should declare "using namespace namespace_name" statement at file scope. Consider the following example.

```
namespace NA
{
        int num1 = 10;
}

using namespace NA;

void show_record(void)
{
        printf("Num1        :       %d\n", num1);
}


void display_record(void)
{
        printf("Num1        :       %d\n", num1);
}
int main(void)
{
        ::show_record();
        ::display_record();
        return 0;
}
```

unnamed or anonymous namespace -

In C/C++,if we declare any global element as static then we can access such element inside same file only. In C++ unnamed namespace or anonymous namespace is superior alternative given for global static member declarations.

```
namespace
{
        int num1;
```

```
}
int main(void)
{
        num1 = 10;
        printf("Num1        :       %d\n", num1);
        return 0;
}
```

Each unnamed namespace has an identifier, assigned and maintained by the program and represented here by unique

above code is implicitly interpreted as follows:

```
namespace unique
{
        int num1;
}
int main(void)
{
        unique::num1 = 10;
        printf("Num1        :       %d\n", unique::num1);
        return 0;
}
```

namespace Alias -

if name of the namespace is too lengthy then we can create alias for it. Consider the following syntax.

```
namespace namespace_name
{
        int num1;
}
namespace NA = namespace_name;
int main(void)
{
        printf("num1 :      %d\n", NA::num1);
        return 0;
}
```

Points To Remember:

1. Namespace is logical term which is used to group or organize functionally equivalent or related code together.

2. In C++, We can achieve logical modularity with the help of namespace.
3. To access members of namespace we should use scope resolution(::)operator.
4. We can write any global variable, function, enum, structure, class as well as namespace inside namespace.
5. We cannot write namespace inside class i.e. A namespace definition must appear either at file scope or immediately within another namespace definition.
6. We cannot implement entry point function(main) inside namespace.
7. Inside same as well as different file we can give same name to the multiple namespace.
8. We cannot create instance of namespace.
9. Instead of declaring global element as static we should put them inside unnamed namespace.

Constructor:

Process of storing user defined value inside variable/object at the time of declaration/creation is called initialization.

e.g. Int number = 123; //initialization

Constructor is member function of a class which is used to initialize the object.

```
namespace NPoint
{
	class Point
	{
	private:
		int xPosition;
		int yPosition;
	public:
		Point( void ) //constructor
		{
			this->xPosition = 0;
			this->yPosition = 0;
		}
	};
}
int main(void)
{
	NPoint::Point pointInstance;
			//Here for pointInstance object constructor will call
```

```
    return 0;
}
```

Due to following reasons constructor is special member function

-its name is same class name.

-it do not have any return type (but we can write return statement inside constructor).

-it is not designed to call explicitly on object rather it gets call implicitly.

-in complete life cycle of an object, constructor gets call only once.

Types of Constructor:

C++ supports three kinds of constructor.

1. Parameterless constructor.
2. Parameterized constructor.
3. Default constructor.

Most of us consider copy constructor is fourth type of constructor. But it is kind of parameterized constructor. We will discuss copy constructor separately.

Parameter less constructor: A constructor which do not take any parameter is called parameter less constructor. parameter less constructor is also called zero parameter constructor or user defined default constructor.

if we create an object without passing argument then parameter less constructor gets call.

Point pt1; Here for pt1 object parameterless constructor will call

Parameterized constructor: If constructor takes parameters then it is called parameterized constructor.

If we create an object by passing argument then parameterized constructor gets call.

Point pt1(10, 20); Here for pt1 object parameterized constructor which is taking 2 parameters will call

Point pt2(500); Here for pt2 object parameterized constructor which is taking 1 parameter will call

How many and which kind of constructor should be exist inside a class depends on instantiation.

Constructor calling sequence is depends on order of object declaration.

Default constructor: If class do not contain any kind of constructor then compiler provides one constructor for the class it is called default constructor.

Default constructor do not perform any operation on data member declared in our class. It look like as follows

```
Point( void )
{
        /*empty*/
};
```

Compiler generated default constructor do not take any parameter i.e. compiler never generate default parameterized constructor. If we want to create an object by passing argument then we cannot rely on default constructor. In this case we should define parameterized constructor inside a class.

We will see exact use of default constructor in virtual function topic.

Constructor is designed for initialization it doesn't mean we cannot call methods from it.

We cannot declare constructor as const, volatile, virtual or static. We can declare constructor as inline only. Inline function we will discuss later.

Aggregate type:

If type allows us to store multiple elements then it is called aggregate type and object is called aggregate object.

In C language, array, structure and union is called aggregate type. Aggregate type allows us to initialize object using initializer list. Consider the following code:

```
int arr[ ] = { 10, 20, 30 };
struct point pt1 = { 10, 20 };
```

class may be classified as aggregate type if it obeys following rule.

1. class should not contain user defined constructor.
2. Class should not contain private and protected members.
3. class should not be derived from any other class.
4. class should not contain virtual function.

Aggregate class is also called Plain Old Data struct(POD- struct).

Test your understanding:

```
class Point
{
private:
        int xPosition;
        int yPosition;
```

```
public:
        Point( void );
        Point( int value );
        Point(int xPosition, int yPosition);
};
```

In following cases which constructor will call?

1. `Point pt1;`

2. `Point pt2( 10 );`

3. `Point pt3( 20, 30 );`

4. `Point pt4( );`

5. `Point pt5 = 40 ;`

6. `Point pt6 = 50, 60;`

7. `Point pt7 = ( 50, 60 );`

8. `Point pt8 = { 70, 80 };`

9. `Point pt9( 90, 100 );`
   `Point pt10 = pt9;`

10. `Point* ptr;`

As per our requirement, we can use any access specifier for constructor. If we declare constructor as private or protected then we can create instance of class only inside member function. If we declare constructor as public then we can create instance of class inside member function as well as non member function.

Note: Generally we should public access specifier for constructor.

Constructors member initializer list:

c++ allows us to initialize data members of the class inside constructors body as well as constructors member initializer list. If we initialize data members inside constructors body

then it gets executed in the same order in which it is written inside constructor body. For example:

```
class Test
{
private:
        int x;
        int y;
        int z;
public:
        Test(void)
        {
                this->z = this->y;
                this->y = this->x;
                this->x = 10;
        }
};
int main(void)
{
        Test test;      //10,garbage value,garbage value
        return 0;
}
```

If we initialize data members inside constructors member initializer list then it gets executed according to the order of data member declaration.

For example:

```
class Test
{
private:
        int x;
        int y;
        int z;
public:
        Test(void) : z(y), x(10), y(x)        //ctor's member initializer list
        {       }
};
int main(void)
{
```

```
Test test;      //10,10,10
return 0; }
```

we cannot initialize array inside constructors member initializer list. It is limitation of constructors member initializer list.

we can use constructors member initializer list and constructor body together. In this case runtime environment executes constructors member initializer list first and then constructors body.

Note:Generally we should use constructor's member initializer to initialize data members. In case of modularity constructors member initializer list must appear in definition part only.

Default argument(s):

```
int sum(int num1, int num2)
{
        return num1 + num2;
}
int sum(int num1, int num2, int num3)
{
        return num1 + num2 + num3;
}
int sum(int num1, int num2, int num3, int num4 )
{
        return num1 + num2 + num3 + num4;
}
int main(void)
{
        int result = 0;
        result = sum(10, 20);
        result = sum(10, 20, 30);
        result = sum(10, 20, 30, 40);
        return 0;
}
```

if we consider above functions then its implementation is logically equivalent. Since all the functions having same name and same type of parameters then we can reuse its implementation by assigning some default values to the parameters. For example

```
int sum(int num1, int num2, int num3 = 0, int num4 = 0 )
{
        return num1 + num2 + num3 + num4;
}
int main(void)
{
        int result = 0;
        result = sum(10, 20);
        result = sum(10, 20, 30);
        result = sum(10, 20, 30, 40);
        return 0;
}
```

Default values assign to the parameters of function is called default argument and function parameter is called optional parameter. In above code 0 is default argument and num3 and num4 are the optional parameters. We cannot assign default arguments randomly. If we want to assign default argument to the parameter then it is necessary to assign default argument to all its right side parameters.

We can assign default arguments to parameters of any member function as well as non member function.

Note:In case of modularity, we must specify default arguments in declarations part only.

Constructor Chaining:

Process of calling constructor from another constructor is called constructor chaining. If we want to reuse implementation of existing constructor then we can use this feature. Since c++ support default argument, it do not support constructor chaining.

In C++ we cannot call constructor with pointer or object explicitly. But it is possible to call to constructor explicitly.

Constant:

l-value(Locator Value):

Expressions that refer to memory locations are called "l-value" expressions. An l-value represents a storage region's "locator" value, or a "left" value, implying that it can appear on the left of the equal sign (=). L-values are often identifiers.

In following example, x is an l-value because it persists beyond the expression that defines it.

```
int main()
{
  int x = 3 + 4;
```

```
    cout << x << endl;
}
```

r-value(Reference Value)

An rvalue is a temporary value that does not persist beyond the expression that uses it. All l-values are r-values but not all r-values are l-values.

In above example, the expression 3 + 4 is an r-value because it evaluates to a temporary value that does not persist beyond the expression that defines it.

Readonly Members:

Once initialized, if we don't want modify state of an object then we should declare variable/object as a constant. In C++ it is compulsory to initialize constant variable.

Note:Generally we should declare function parameter as a constant.

Constant Data Member:

if we dont want to modify state of the data member inside any member function of the class(including contructor) then we should declare data member as a constant.

Note:It is compulsory to initialize constant data member inside constructors member initializer list.

```
class Math
{
private:
        const double PI;      //constant data member
public:
        Math(void) : PI(3.14)
        {
                //this->PI = 3.142;          //Not allowed
        }
};
```

Constant Member function:

If we don't want to modify state on current object inside member function then we should declare member function as constant. We can declare only such function as a constant which get this pointer. Since global function do not get this pointer, we cannot declare global function as a constant.

Non constant member function always get this pointer of following type:

ClassName* const this;

Constant member function always get this pointer of following type:

const ClassName* const this;

As shown in above declaration,if we declare member function as constant then current object is considered as const object
inside that member function.

```
class Test
{
private:
	int number;
public:
	Test(void) : number(10)
	{	}
	//Test* const this = &testInstance;
	void increment()
	{
		this->number = this->number + 1;	//allowed
	}
	//const Test* const this = &testInstance;
	int getNumber(void)const
	{
		//this->number = this->number + 1;  //Not allowed
		return this->number;
	}
};
int main(void)
{
	Test testInstance;
	return 0;
}
```

With the help of non constant object we can invoke constant as well as non constant member function.

mutable is a keyword in c++, which allows us to modify state of non constant data member inside constant member function.

```
class Test
```

```
{
private:
        const int num1;
        const int num2;
        mutable int count;
public:
        Test(const int num1 = 0, const int num2 = 0) : num1(num1), num2(num2)
        {
                this->count = 0;
        }
        void print(void)const
        {
                this->count = this->count + 1; //allowed
                cout << "Count    :       " << this->count << endl;
                cout << "Num1     :       " << this->num1 << endl;
                cout << "Num2     :       " << this->num2 << endl;
        }
};
int main(void)
{
        Test testInstance;
        testInstance.print();
        testInstance.print();
        return 0;
}
```

We can declare object of class as a constant. Using constant object we can call only constant member function.

const, volatile and mutable is called Access modifier.

Reference:

typdef is feature of C language which allows us to create alias for the existing data type.

e.g. typedef unsigned int size_t;
size_t size = sizeof(int);

Reference is feature of C++ which allows to create alias for existing object.

e.g. int num1 = 10;

```
int& num2 = num1;
```

In above code, num2 is reference variable where num1 is referent variable. Once initialized we cannot change referent of reference.

```
int num1 = 10;
int num2 = 20;
int& num3 = num1;       //num3 is reference variable for num1;
num3 = num2;            //Here we are not changing reference ,ather copy value from
```

num2 into num3;

Since references are implicitly considered as a constant pointer, it is compulsory to initialize references. Following code illustrate how references are implicitly const and pointer.

```
class Test
{
private:
    double& num3;
public:
    //It is compulsory to initialize references in ctors member initilaizer list
    Test(double& num2) : num3( num2 )
    {    }
};
int main(void)
{
    double num1 = 10;
    Test testInstance( num1 );
    size_t size = sizeof(testInstance);      // 4 bytes
    return 0;
}
```

Even though data member is a type of double, size of an object is 4 bytes, it means that references are implicitly treated as a pointer. References need to initialize inside ctors member initializer list it means that references are implicitly treated as const.

```
int num1 = 10;
int& num2 = num1;
```

Now we can treat above statement as follows:

```
int num1 = 10;
```

```
int& num2 = num1;        //int* const num2 = &num1;
```

i.e. References are automatically dereferenced constant pointer variables.

In following code i have used constant in combination with reference.

```
int num1 = 10;
int& num2 = num1;
const int& num3 = num1;
```

Here we can read as well as modify state of num1 via num2 but we cannot modify state via num3.

References are not designed to use independently rather it is designed to pass and return object from function by reference.

Note:Generally we should not return local variable by reference. In this case we should declare local variable as static.

Points To Remember:

1.Reference is derived data type.
2.Reference is used to create alias for the object.
3.We cannot change referent of the reference.
4.We cannot create reference to reference.
5.We cannot initialize reference to NULL.
6.We cannot create reference to constant.
7.We cannot create array of references.
8.We can create reference to pointer, array, function as well as object.
9.We can pass object to the function by value, by address and by reference.
10.With the help of reference we cannot save the memory rather we can minimize complexities of pointer.

Test your understanding:

How will you create reference to array?
What is the difference between pointer and reference?

Macro vs Inline Function:

Macro: A macro is short form of macro instruction.

It is a rule or pattern that specifies how a certain sequence of characters should be mapped to a replacement output sequence of characters according to a defined procedure. In short,

symbolic constant is also called macro. The mapping process that transforms a macro use into a specific sequence is known as macro expansion. Preprocessor is program which expands the macro.

Preprocessor performs two main task:

1.Expands the macro and

2.Removes the comments which included in .c file

There are two types of macros, object-like and function-like. Object-like macros do not take parameters; function-like macros do.

Consider the example of object like macro

```
#define     PI      3.142
```

Consider the example of function like macro

```
#define SQUARE( number )        number * number
int main(void)
{
        int number = 5;
        cout << SQUARE(number) << endl;
        return 0;
}
```

Several identifiers are predefined, and expand to produce special information.

__LINE__ A decimal constant containing the current source line number.

__FILE__ A string literal containing the name of the file being compiled.

__DATE__ A string literal containing the date of compilation, in the form "Mmmm dd yyyy"

__TIME__ A string literal containing the time of compilation, in the form "hh:mm:ss"

Inline Function:

Giving call to the function is always overhead to the compiler because each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time.

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. Instead of transferring control to and from the function code segment, a modified

copy of the function body may be substituted directly for the function call. In this way, the performance overhead of a function call is avoided.

To declare function as inline we should use inline keyword.

```
inline int max( int num1, int num2 )
{
	return num1 > num2 ? num1 : num2;
}
```

If we implement member function inside structure/class then it is by default treated as inline.If we define functions globaly then we must specify inline keyword explicitly. Consider the following code.

```
class Point
{
private:
	int xPosition;
	int yPosition;
public:
	inline Point();
	inline void printRecord();
};
inline Point::Point()
{
	this->xPosition = 0;
	this->yPosition = 0;
}
inline void Point::printRecord()
{
	cout << "X Position	:	" << this->xPosition << endl;
	cout << "Y Position	:	" << this->yPosition << endl;
}
int main(void)
{
	Point pt1;
	pt1.printRecord();
	return 0;
}
```

We can declare constructor as inline only.

The inline specifier is only a suggestion/request to the compiler that an inline expansion can be performed; the compiler is free to ignore the suggestion.

In following condition, function cannot be classified as inline
1.If function contains a code which leads to larger executables.
2.If we implement function using recursion.
3.If we implement function using loop.

Note:If we declare function as inline then we cannot divide class declaration and definition in multiple files.
Test your understanding:
Why inline functions are hazardous?

string class:
C as well as C++ programming language supports strings but string is not a fundamental data type in it.Standard Template Library contains class 'basic_string' which allows us to handle the string. string is typedef of basic_string.

typedef basic_string string.

'basic_string' class is declared in string(#include<string>)header file. Lets see how to use it.

```
string name;
cout<<"Enter your name :       "';
cin>>name;
cout<<"Name      :       "<<name<<endl;
```

Now no need to declare character array. In MS Visual Studio size of object os string class is ____ bytes.

Exception Handling:

'An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions'.

To create an application, most of the times we take help of operating system resources. Following are some the resources we use frequently:

1. Memory
2. File
3. Socket etc.

Since these resources are very limited we should use it carefully. To manage these resources efficiently, we should use exception handling. In C++ we can handle exceptions with the help of following keywords:

1.try
2.throw
3.catch

if we want to inspect group of statements for exception then we should put such statement inside try block or try handler. If we want to use try block then it is necessary to provide at least one catch block.

Exceptions may be generated implicitly by the C++ runtime or programmer can generate it explicitly with the help of throw keyword. throw is jump statement. We can use throw statement inside try as well as catch block.

To handle exception thrown from try block we should use catch block or catch handler. Single try block may have multiple catch block but it must have at least one catch block.

```
void accept_record(int& number)
{
        cout << "Enter number   :       ";
        cin >> number;
}
void print_result(const int& result)
{
        cout << "Result     :       " << result << endl;
}
int main(void)
{
```

```
try
{
        int num1;
        ::accept_record(num1);

        int num2;
        ::accept_record(num2);

        if (num2 == 0)
                throw string("Divide By Zero Exception");
        int result = num1 / num2;
        cout << "Result     :       " << result << endl;
}
catch ( string ex )
{
        cout << ex << endl;
}
return 0;
}
```

We can write try catch block inside another try as well as catch block. Outer catch block can handle exception which are unhand-led by the inner catch block but catch block of inner try cannot handle exception which are unhand-led by outer catch block.

If we do not handle thrown exception then C++ runtime implicitly gives call to 'std::terminate' function which implicitly gives call to the 'std::abort' function which stops execution of program abnormally.In this case we should write such a catch block which can handle all kind of exception thrown by user. A catch block which can handle all types of exception is called generic catch block.

```
catch( ... )
{
        /* Inside generic catch block*/
}
```

If we want to use generic as well as specific catch handler together then generic catch handler must be the last handler.

In C++, we can throw exception from outside try block via function. Consider the following example:

```
void accept_record(int& number)
```

```
{
        cout << "Enter number    :       ";
        cin >> number;
}
int calculate(int num1, int num2)
{
        if (num2 == 0)
                throw string("Divide By Zero Exception");
        return num1 / num2;
}
void print_result(const int& result)
{
        cout << "Result     :       " << result << endl;
}
int main(void)
{
        try
        {
                try
                {
                        int num1;
                        ::accept_record(num1);
                        int num2;
                        ::accept_record(num2);
                        int result = ::calculate(num1, num2);
                        ::print_result(result);
                }
                catch (string ex)
                {
                        cout << ex << endl;
                }
        }
        catch (...)
        {
                cout << "Exception" << endl;
        }
```

```
    return 0;
}
```

In case of modularity, We can intimate to the user to handle specific exceptions with the help of exception specification list. Consider the following code

```
int calculate(int num1, int num2) throw ( string )
{
    if (num2 == 0)
        throw string("Divide By Zero Exception");
    return num1 / num2;
}
```

throw followed by type of exception is called exception specification list.

Note:We can write exception specification list inside declaration as well definition but ideally we should write it in declaration part only.

If we forget to specify type of exception in exception specification list then catch block do not handle such exception rather C++ runtime implicitly gives call to the std::unexpected function which gives call to the std:: terminate function. This feature do notify works in Microsoft Visual Studio. To test this feature we should write code in unix/linux.

If information required to handle the exception is incomplete at given place then we can rethrow the exception to its outer catch using throw keyword.

```
    try
    {
        try
        {
            int num1;
            ::accept_record(num1);
            int num2;
            ::accept_record(num2);
            int result = ::calculate(num1, num2);
            ::print_result(result);
        }
        catch (string ex)
        {
            throw;      //rethrow
        }
    }
```

```
        catch (string ex)
        {
                cout << ex << endl;
        }
        catch (...)
        {
                cout << "Exception" << endl;
        }
```

Sometimes we need to handle the exception by throwing different type of exception. It is called exception chaining. Consider the following example:

```
class LinkedList
{
        Node* head;
public:
        bool empty(void)
        {
                //TODO
        }
        void deleteFirst(void)throw( string )
        {
                if (this->empty())
                        throw string("List is empty");
                else
                        //TODO
        }
};
class Stack
{
private:
        LinkedList collection;
public:
        void pop(void)throw( string )
        {
                try
                {
                        collection.deleteFirst();
```

```
            }
            catch (string e)
            {
                  throw string("Stack is full");       //Exception chaining
            }
      }
};
```

Stack Unwinding:

In the C++ exception mechanism, control moves from the throw statement to the first catch statement that can handle the thrown type. When the catch statement is reached, all of the automatic variables that are in scope between the throw and catch statements are destroyed in a process that is known as stack unwinding.

In stack unwinding, execution proceeds as follows:

1.Control reaches the try statement by normal sequential execution. The guarded section in the try block is executed.

2.If no exception is thrown during execution of the guarded section, the catch clauses that follow the try block are not executed. Execution continues at the statement after the last catch clause that follows the associated try block.

3.If a matching handler is still not found, or if an exception occurs during the unwinding process but before the handler gets control, the predefined run-time function terminate is called. If an exception occurs after the exception is thrown but before the unwind begins, terminate is called.

4.If a matching catch handler is found the process of unwinding stack begins. This involves the destruction of all automatic objects that were fully constructed—but not yet destructed—between the beginning of the try block that is associated with the catch handler and the throw site of the exception. Destruction occurs in reverse order of construction. The catch handler is executed and the program resumes execution after the last handler—that is, at the first statement or construct that is not a catch handler.

Types of Function:

An operation denotes a service that a class offers to its clients. Client typically performs five kinds of operations on an object.These functions allows us to write loosely coupled class.

1. Selector: an operation that accesses the state of an object but does not alter the state. Selector function is also called inspector function.

```
int getReal( void )const
{
        return this->_real;
}
```

Generally we should declare inspector function as a constant.

2.Modifier: an operation that alters the state of an object. Modifier function is also called mutator function.

```
void setReal( const int real )
{
        this->_real = real;
}
```

3. Iterator: an operation that permits all parts of an object to be accessed in some well-defined order.

4. Constructor: an operation that initializes an object.

5. Destructor: an operation that deinitializes an object.

Header Guard:

In case of repeated inclusion of headers, if we want to replace contents of header only once then we should use header guard.Header guard is also called #include guard or macro guard. Header guards are implemented by using three preprocessor directives in a header file. Two are placed at the beginning of the file, before any pertinent code. The last is placed at the end of the file.

```
#ifndef unique_symbol
#define unique_symbol


#endif
```

The symbol used is not crucial, but it must be unique. It is traditional to use all capital letters for the symbol. Only letters, numbers and the underscore character can be used in the symbols. No other punctuation is allowed. A very common symbol is to use the name of the header file, converting the .h suffix to a _H.

Consider the Complex.h header file with header guard as follows:

```
#ifndef     COMPLEX_H
#define     COMPLEX_H

    class Complex
    {
    private:
        int real;
        int imag;
    public:
        int getReal()const;
        void setReal( const int real );
        ...
        ...
    };

#endif
```

Modular approach:

If we want to divide code into multiple files then class declaration should appear in header file and member function definition should appear in source file. We should define all global functions in Main.cpp file.

To define functions globally, we should use following format:

```
ReturnType ClassName::MemberFunctionName( ... )
```

Points To Remember:

1.We must specify default arguments only in declaration part only(.h file).
2.Constructors member initializer list must be appear in definition part only(.cpp).
3.const keyword must be appear in declaration as well as definition part(.h & .cpp).
4.static keyword must be appear in declaration part only(.h file).
5.Global definition for static data member must be appear in source file(.cpp).
6.Friend keyword must be appear in declaration part only.(.h)
7.virtual keyword must be appear in declaration part only(.h file).

8.We should not define pure virtual function inside source file.

9.Exception specification list can be appeared in declaration as well as definition. But it must appear in declaration.

10.User defined header files should be included using double quotes(#include"Complex.h").

11.In case of multiple files,we should use following sequence of header files.

i.all C language standard header files.

ii.all C++ language standard header files.

iii.all user defined header files.

Modularity:

-Process of dividing complex system into modules which can be compiled separately, but which have connections with other modules is called modularity.

-It is major pillar of oops.

-Modularity helps to minimize module dependancy.

-In C++, we can achieve logical modularity with the help of namespace and physical modularity with the help of multiple files.

Abstraction:

-Process of getting essential characteristics of an object is called abstraction.

-An abstraction focuses on the outside view of an object.

-Abstraction focuses on the essential characteristics of some object,relative to the perspective of the viewer.

kinds of abstractions include the following:

Entity abstraction =>An object that represents a useful model of a problem domain or solution domain entity

Action abstraction =>An object that provides a generalized set of operations, all of which perform the same kind of function.

Virtual machine abstraction => An object that groups operations that are all used by some superior level of control, or

operations that all use some junior-level set of operations

Coincidental abstraction => An object that packages a set of operations that have no relation to each other

Encapsulation:

-Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.

-Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior or binding of data and code together is called encapsulation.

-Encapsulation hides the details of the implementation of an object.

Information Hiding :

It is the process of hiding all the secrets of an object that do not contribute to its essential characteristics. Generally we can hide data/information with the help of private or protected access specifier.

Data Security.

Process of giving control access to the members of class is called data security. To achieve data security first encapsulate variable and function inside a class and then the help of access specifiers give access to the data members using getter and setter method.

```
class Account
{
private:
        float balance;
public:
        void withdraw( float amount )
        {
                if( amount >= 100 )
                        this->_balance = this->balance - amount;
                else
                        throw Exception("Invalid amount");
        }
}
```

Function Overloading:

If implementation of function is logically equivalent then we should give same name to the function. If we want to give same name to the function then we must obey following rules:

1.number of parameters pass to the function must be different.

```
void sum( int num1, int num2 ){         }
void sum( int num1, int num2, int num3 ){    }       //allowed
```

2.If number of parameters are same then type of at least one parameter must be different.

```
void sum( int num1, int num2 ){         }
void sum( int num1, double num2 ){   }       //allowed
```

3.If number of parameters are same then order of type of parameter must be different.

```
void sum( int num1, float num2 ){      }
void sum( float num1, int num2 ){      }      //allowed
```

4.On the basis of return type we cannot give same name to the function. i.e. return type is not considered.

```
void sum( int num1, int num2 ){        }
int sum( int num1, int num2 ){  }      //Not allowed
```

Process of writing such functions which is having same name but different signature is called function overloading.Functions which are taking part into function overloading are called overloded function. In case of overloading functions must be exist inside same scope. Except main function and destructor, we can overload any global function as well as member function.

Since ANSI has not defined any strict specification on return type, catching values from the function are optional. so return type is not considered at the time of overloading.

Test Your Understanding

1.What will be the output of following program?

```
void print(int number)
{
        cout << "int :        " << number << endl;
}
void print(double number)
{
        cout << "double    :        " << number << endl;
}
1.      print( 10 );

2.      print( 10.5 );

3.      print( 10.5f );
```

2.What will be the output of following program?

```
void print(int number)
{
```

```
        cout << "int :       " << number << endl;
}
void print(float number)
{
        cout << "float       :       " << number << endl;
}
1.      print( 10 );

2.      print( 10.5 );

3.      print( 10.5f );
```

3.What will be the output of following program?

```
void print(int number)
{
        cout << "int :       " << number << endl;
}
void print(int& number)
{
        cout << "int&        :       " << number << endl;
}
1.      print( 10 );

2.      int number = 10;
        print( number );
```

4.What will be the output of following program?

```
void print(bool object)
{
        cout << "bool        :       " << object << endl;
}
void print(char object)
{
        cout << "char        :       " << object << endl;
}
```

```
1.      print( true );

2.      print( 'A' );

3.      print( 65 );
```

5.What will be the output of following program?

```
void print(int num1 = 0)
{
        cout << "int :       " << num1 << endl;
}
void print(int num1, int num2 = 0)
{
        cout << "int :       " << num1 << endl;
        cout << "int :       " << num2 << endl;
}
1.      print(0, 10);

2.      print(30);

3.      print(10, 20 );

4.      print();
```

6.What will be the output of following program?

```
void print( char text[])
{
        cout << "char[]    :       " << text << endl;
}
void print(char* text)
{
        cout << "char*     :       " << text << endl;
}
1.      print("Sandeep");

2.      char name[] = "Sandeep";
```

```
print(name);
```

7.What will be the output of following program?

```
void print( const char* text)
{
        cout << "const char*      :       " << text << endl;
}
void print(char* const text)
{
        cout << "char* const      :       " << text << endl;
}
1.      print("Sandeep");

2.      char name1[] = "Sandeep";
        print(name1);

3.      const char name2[] = "Sandeep";
        print(name2);
```

8.What will be the output of following program?

```
class Test
{
public:
        void Print( void )
        {
                cout << "void print( void )" << endl;
        }
        void Print(void)const
        {
                cout << "void print( void )const" << endl;
        }
};
1.      Test t1;
        t1.Print();

2.      const Test t2;
```

```
t2.Print();
```

9.What will be the output of following program?

```
enum E
{
        eNumber = 10
};
void print(const int number)
{
        cout << "Number  :       " << number << endl;
}
void print(E number)
{
        cout << "Number  :       " << number << endl;
}
1.      print(10);

2.      const int number = 10;
        print(number);

3.      print(eNumber);
```

Name Mangling and Mangled Name:

C++ compiler generates encoded name by looking name of the function and type of argument pass to the function is called mangled name.

```
        void sum( int num1, int num2 );                 //sum@@int,int

        double sum( int num1, float num2, double num3);         //sum@@int,float,double
```

Since ANSI has not defined any specification on mangled name, compiler vendors are free to generate mangled name.It means that mangled name may vary from compiler vendor to compiler vendor.It you try to access any element without definition, compiler will return you mangled name.

Process or algorithm which decides how to generate mangled name is called name mangling. Generating mangled name is job of compiler.

If we want to access functions implemented in .c file into .cpp then it is necessary to declare functions as extern "C".

For functions declared as extern "C", compiler do not generate mangled name.

Dynamic Memory allocation:

C as well as C++ language allows us to reserve space for an object statically as well as dynamically. First we will see how to reserve space using C and then C++.

To allocate and deallocate space for an object dynamically we should use following functions. To use these functions we should use stdlib.h (cstdlib in C++) header file.

1. void* malloc(size_t size);

- The malloc() function allocates size bytes and returns a pointer to the allocated memory.
- The memory is not initialized.
- If size is 0, then malloc() returns either NULL, or a unique pointer value.

2.**void* calloc(size_t nmemb, size_t size);**

- The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory.
- The memory is set to zero.
- If nmemb or size is 0, then calloc() returns either NULL, or a unique pointer value.

3.void* realloc(void * ptr, size_t size);

- The realloc() function changes the size of the memory block pointed to by ptr to size bytes.
- The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.
- If the new size is larger than the old size, the added memory will not be initialized.
- If ptr is NULL, then the call is equivalent to malloc(size).
- if size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).
- If there is not enough available memory to expand the block to the given size, the original block is left unchanged, and NULL is returned.
- If the area pointed to was moved, a free(ptr) is done.

4.void free(void * ptr);

- The free() function deallocates the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc().

- If ptr is NULL, no operation is performed.

Standard directory for visual studio(C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include) contains new.h (and new also) header file. It contains declarations of following functions:

```
struct nothrow_t{  // Empty structure };

extern const nothrow_t nothrow;          // constant for placement new tag

void* operator new(size_t size) throw(bad_alloc);

void operator delete(void *) throw();

void* operator new[](size_t _Size)throw(bad_alloc);          // allocate array or throw exception

void  operator delete[](void *) throw();          // delete allocated array

void* operator new(size_t size, void *where) throw();

void* operator new(size_t _Size, const nothrow_t&) throw();

void*  operator new[](size_t _Size, const  nothrow_t&)throw(); //      allocate  array  or  return null pointer
```

In C++, to allocate memory dynamically we should use new operator and to deallocate that memory we should use delete operator. new implicitly calls operator new and delete implicitly calls operator delete function which are declared in new.h header file. Lets see how to use new and delete operator.

1. Allocating and deallocating space dynamically for single integer varaible:

```
try
{
        int* ptr  = new int;
        //int* ptr = ( int* )::operator new( sizeof( int ) * 1 );
        ...
        ...
```

```
        delete ptr;             //::operator delete( ptr );
        ptr = NULL;
    }
    catch( bad_alloc ex )
    {
        cout<<ex.what()<<endl;
    }
```

2.Allocating and deallocating space dynamically for single dimensional array of integer:

```
    try
    {
        int* ptr = new int[ 3 ];
        //int* ptr = ( int* )::operator new[]( sizeof( int ) * 3 );
        ...
        ...
        delete[] ptr;                   //::operator delete[]( ptr );
        ptr = NULL;
    }
    catch( bad_alloc ex )
    {
        cout<<ex.what()<<endl;
    }
```

3.Allocating and deallocating space dynamically for multidimensional array of integer:

```
    try
    {
        int** ptr = new int*[ 2 ];
    //int** ptr = ( int** )operator new[]( sizeof( int* ) * 2 );

    for( int index = 0; index < 2; ++ index )
        ptr[ index ] = new int[ 3 ];
    //ptr[ index ] = ( int* )::operator new[]( sizeof( int ) * 3 );
        .....
        .....
        for( int index = 0; index < 2; ++ index )
            delete[] ptr[ index ];          //::operator delete( ptr[ index ] );
```

```
        delete[] ptr;                   //::operator delete( ptr );
        ptr = NULL;
    }
    catch( bad_alloc ex )
    {
        cout<<ex.what()<<endl;
    }
```

If new operator fails to allocate memory then C++ runtime implicitly throws bad_alloc exception.

We can allocate and deallocate memory for an object of user defined data type. In this case new implicitly calls constructor and delete implicitly calls destructor.

```
1. Point* ptr = new Point;          //parameterless constructor will call
2. Point* ptr = new Point();        //parameterless constructor will call
//Recommended
3. Point* ptr = new Point( 10, 20 );    //parameterized constructor will call
4. delete ptr;                  //destructor will call
```

Main advantage of new operator over malloc function is that it allows us to reserve space for an object on allocated as well as un-allocated memory area. To reserve space on allocated space we should use following form of new operator. It is called placement new operator.

Consider the example;

```
    char buffer[ 30 ];
    int* ptr = new ( buffer ) int;
//int* ptr = ( int* )::operator new( sizeof( int ) * 1 , buffer );
    ....
    ....
    //delete ptr; //Not allowed
    ptr = NULL;
```

if we want to place an object at a particular location in memory then we can use placement new operator.

If malloc fails to allocate memory then it returns NULL. new operator may behave like this if we use nothrow object.

consider the following example:

1. Following code demonstrate bad_alloc exception.

```
    try
    {
```

```
        int size = 1000000000;
        int* ptr = new int[ size ];
    }
    catch (bad_alloc ex)
    {
        cout << ex.what();
    }
    //Output    :    bad allocation
```

2. Following code demonstrate use of nothrow.

```
    try
    {
        int size = 1000000000;
        int*  ptr  =  new  (nothrow)int[size];  //int*  ptr  =  (int*)::operator
new[](sizeof(int)* size, nothrow);
        if (ptr == NULL)
            cout << "NULL" << endl;
    }
    catch (bad_alloc ex)
    {
        cout << ex.what();
    }
    //Output    :    NULL
```

nothrow_t is an empty structure which is declared in new.h header file and nothrow is an instance of nothrow_t struct.

Wild pointer: Uninitialized pointer is called wild pointer.

```
    int* ptr;    //wild pointer
```

Dangling pointer : A pointer which stores address of released memory or pointer which points to the invalid memory location is called dangling pointer;

```
    int* ptr1 = new int();    //default value is 0
    int* ptr2 = ptr1;    //two pointers for same location
    *ptr2 = 10;
    cout << "Value    :    " << *ptr2 << endl;
    delete ptr2;
```

```
//Now ptr1 becomes dangling pointer
cout << "Value      :        " << *ptr2 << endl;
delete ptr1;  //Not allowed
```

To avoid dangling pointer we should deinitialize all pointers which are pointing to same location to NULL.

Memory leakage: If we reserve space in memory but no pointer is pointing to it then wastage of such memory is called memory leakage.

```
int* ptr = new int[3];
...
ptr = new int[5];    //this statement definitely loose 12 bytes.
```

To avoid memory leakage either we should use delete operator properly or we should use smart pointer.

Smart Pointer : If we treat an object as pointer then such object is called smart pointer. auto_ptr and unique_ptr are the predifined smart pointers. We will discuss smart pointer later. Consider the following example:

```
#include<iostream>
#include<memory>
using namespace std;
int main(void)
{
        auto_ptr<int> autoPointer(new int());

        *autoPointer = 10;

        cout << "Value      :        " << *autoPointer << endl;

        return 0;
}
```

Test your understanding:

1. int* ptr = new int( );
2. int* ptr = new int(3);
3. int* ptr = new int[3];

What is the difference between malloc and new?

Destructor:

A member function of class that has the same name as the class and is prepended with ~ operator is called destructor. It is used to do clean up after an object of a class is no longer in the scope of program segment. Consider the syntax for destructor:

```
~Point( void )        //destructor for the Point class
{


}
```

There are two situations in which a destructor is called. The first is when an object is destroyed under "normal" conditions, e.g., when it goes out of scope or is explicitly deleted. The second is when an object is destroyed by the exception-handling mechanism during the stack-unwinding part of exception propagation.

If we do not define destructor for the class then compiler provides one destructor for the class by default. It is called default destrucotr. Default destructor always does the job of deinitialiation only.

On exit from function or block, Objects created in that function are destroyed by the compiler automatically. But an object might have an accumulated resources (dynamic memory, disk blocks, network connections etc.)during its life time. These resources are stored in the objects data members and are manipulated by the member functions. When the object destroyed, it is necessary that the resources held by the object are released because object is no longer accessible. Because compiler isn't aware of the resources held by the object, it is responsibility of object to release such resources. To help an object in doing so, on exit from function, destructor is called on all objects created statically. In other words, to release operating system resources held by the object we should write destructor inside class. Consider the following code snippet:

```
namespace NString
{
        class String
        {
        private:
                unsigned int length;
                char* buffer;
```

```
    public:
        String(unsigned int length = 0 )//ctor
        {
            //TODO
        }
        ~String(void)//dtor
        {
            if (this->buffer != NULL)
            {
                delete[] this->buffer;
                this->buffer = NULL;
            }
        }
    };
}
```

Points To Remember:

1.To Prevent resource leak we should use destructor.

2.Destructor calling sequence is exactly opposite of constructor calling sequence(in array too).

3.Since destructor do not take any parameter we cannot overload it.

4.We can declare destructor as inline and virtual only.

5.Even though destructor is designed to call implicitly, we can call it explicitly too.

6.if we try to create object using new then constructor gets call and if we try to delete that object using delete
operator then destructor gets call. Consider following example:

```
String* ptr = new String( );        //Here ctor will call implicitly
...
delete ptr;                         //Here dtor will call implicitly
```

If we create an instance with the help of placement new operator then we cannot deallocate memory of an object using delete operator. If we do not use delete operator then destructor will not be call. But to avoid resource leak we need to call destructor explicitly. Consider the following code.

```
char arr[30];
```

What is the difference between malloc and new?

Destructor:

A member function of class that has the same name as the class and is prepended with ~ operator is called destructor. It is used to do clean up after an object of a class is no longer in the scope of program segment. Consider the syntax for destructor:

```
~Point( void )        //destructor for the Point class
{


}
```

There are two situations in which a destructor is called. The first is when an object is destroyed under "normal" conditions, e.g., when it goes out of scope or is explicitly deleted. The second is when an object is destroyed by the exception-handling mechanism during the stack-unwinding part of exception propagation.

If we do not define destructor for the class then compiler provides one destructor for the class by default. It is called default destrucotr. Default destructor always does the job of deinitialiation only.

On exit from function or block, Objects created in that function are destroyed by the compiler automatically. But an object might have an accumulated resources (dynamic memory, disk blocks, network connections etc.)during its life time. These resources are stored in the objects data members and are manipulated by the member functions. When the object is destroyed, it is necessary that the resources held by the object are released because object is no longer accessible. Because compiler isn't aware of the resources held by the object, it is responsibility of object to release such resources. To help an object in doing so, on exit from function, destructor is called on all objects created statically. In other words, to release operating system resources held by the object we should write destructor inside class. Consider the following code snippet:

```
namespace NString
{
        class String
        {
        private:
                unsigned int length;
                char* buffer;
```

```
    public:
        String(unsigned int length = 0 )//ctor
        {
            //TODO
        }
        ~String(void)//dtor
        {
            if (this->buffer != NULL)
            {
                delete[] this->buffer;
                this->buffer = NULL;
            }
        }
    };
}
```

Points To Remember:

1.To Prevent resource leak we should use destructor.

2.Destructor calling sequence is exactly opposite of constructor calling sequence(in array too).

3.Since destructor do not take any parameter we cannot overload it.

4.We can declare destructor as inline and virtual only.

5.Even though destructor is designed to call implicitly, we can call it explicitly too.

6.if we try to create object using new then constructor gets call and if we try to delete that object using delete
operator then destructor gets call. Consider following example:

```
String* ptr = new String( );        //Here ctor will call implicitly
...
delete ptr;                         //Here dtor will call implicitly
```

If we create an instance with the help of placement new operator then we cannot deallocate memory of an object using delete operator. If we do not use delete operator then destructor will not be call. But to avoid resource leak we need to call destructor explicitly. Consider the following code.

```
char arr[30];
```

```
String* ptr = new ( arr )String();
....
....
ptr->~String();       //explicit call to dtor
ptr = NULL;
```

Array of Objects:

```
Point** ptr = new Point*[3];       //A
for (int index = 0; index < 3; ++index)
        ptr[index] = new Point();


.....
.....
for (int index = 0; index < 3; ++index)
        delete ptr[index];
delete[] ptr;
ptr = NULL;
```

Object Copy Semantics:

There are three different strategies for object coping.

1. Shallow Copy. (Also known as bitwise copy)
2. Deep Copy. (Also known as member-wise copy)
3. Lazy Copy. (Also known as Copy On Write(COW).)

The term shallow,deep & lazy copy originated in smalltalk and these terms are generally used to describe copy semantics

Shallow Copy:

At the time of copy operation, if we copy every member of source object into destination object as it is then such type of copy is shallow copy. Consider the following example:

1.
```
int num1 = 10;
int num2 = num1; //shallow copy.
```

2.
```
double num1 = 20.5;
double num2;
num2 = num1;              //shallow copy.
```

3. Point pt1(10, 20);
   Point pt2;
   pt2 = pt1; //shallow copy.

4. String str1("Sandeep");
   String str2 = str1; //shallow copy

Deep Copy:

If class data member holds operating system resource & if we want to create copy of the object then we need to rely on deep copy operation. Here i am going to consider memory for discussing deep copy.

If class contains at least one data member of pointer type, class contains user defined destructor which does job of deallocation and if we try to copy object into another object then instead of coping memory address from data member of source object into data member of destination object, we should allocate new memory for the data member of destination object and then we should copy contents from memory of source object into memory of destination object. Such type of copy is called deep copy.

Conditions to create deep copy.

1.Class should contain at least one data member of pointer type.
2.Class contains user defined destructor which does job of deallocation
3.We should create copy of object

Copy of the object gets created due to following reasons:

1.Initialization
2.Assignment
3.Passing object to the function by value.
4.Returning object function by value.

Steps to create deep copy:

1.Copy the size/length from data member of source object into data member of destination object.
2.Allocate new memory for the pointer data member of destination object.
3.Copy the contents from memory of source object into memory of destination object.

Where to create deep copy.:

1.In case of assignment we should create deep copy inside assignment operator function.[Operator Overloading]

2.In rest of the condition we should create deep copy inside copy constructor.

Copy Constructor:

A member function of class having same name as a class and which takes only one parameter of same type but as a reference is called copy constructor of the class. Copy constructor is a kind of parameterized constructor(Single Parameter Constructor).

General Syntax of copy constructor is as follows:

```
ClassName( const ClassName& other )
{
        //TODO:Copy operation
}
```

Initialization is the process of storing known value in a variable during its creation. Job of copy constructor is to Initialize newly created object from existing object.

The copy constructor gets invoked in following conditions:

1.If we pass an object to the function by value.

```
class Point
{
private:
        int xPosition;
        int yPosition;
public:
        //Point* const this = & pt1;
        //Point pointInstance = pt2;
        void sum(Point pointInstance)   //Here for pointInstance, copy constructor will call
        {
                //TODO
        }
};
int main(void)
{
        Point pt1(10, 20);  //for pt1 parameterized ctor will call
```

```
        Point pt2(30, 40);  //for pt2 parameterized ctor will call
        pt1.sum(pt2);
        return 0;
    }
```

2.If we return an object from function by value.

```
    class Point
    {
    private:
        int xPosition;
        int yPosition;
    public:
        //Point* const this = &pt1;
        //Point pointInstance = pt2;
        Point sum(Point pointInstance)
        {
            Point tempInstance;
                //TODO
            return tempInstance;
        }
    };
    int main(void)
    {
        Point pt1(10, 20);  //for pt1 parameterized ctor will call
        Point pt2(30, 40);  //for pt2 parameterized ctor will call
        Point pt3;          //for pt3 parameterless ctor will call
        pt3 = pt1.sum(pt2);//Here First Sum function & then for annonymous object copy constructor will call
        return 0;
    }
```

3.If we initialize newly created object from existing object.

```
    String str1("Sandeep");
    String str2 = str1; //Here for str2 copy constructor will call
```

or

```
    String str1("Sandeep");
```

String str2(str1);//Here for str2 copy constructor will call

4. If we throw and catch object of user defined type.

```
class Exception
{
private:
        int length;
        char* text;
};

int main(void)
{
        int num1 = 10;
        int num2 = 0;
        try
        {
                if (num2 == 0)
                        throw Exception("Divide By Zero Exception");
                int res = num1 / num2;
        }
        catch (Exception ex)        //Here for ex copy constructor will call
        {
                cout << "Exception" << endl;
        }
        return 0;
}
```

Note:It is however, not guaranteed that a copy constructor will be called in these cases because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the return value optimization

If class do not contain copy constructor then compiler provides one copy constructor for the class. Such copy constructor is called default copy constructor. Default behavior of default copy constructor always creates shallow copy of an object. In case of initialization of an

object, if there is need to create deep copy then we should define user defined copy constructor inside a class.

```
class String
{
private:
        int length;
        char* buffer;
public:
        String( const String& other ) throw( bad_alloc )       //copy constructor with deep
copy
        {
                this->length = other.length;
                this->buffer = new char[ this->length + 1 ];
                strcpy( this->buffer, other.buffer );
        }
};
```

If we pass an object to the function by address or by reference then copy constructor do not call.

Lazy Copy:

String class which is declared above is fairly easy to understand and implement. What happens when objects of this class are used heavily as a function arguments and as a return values using the pass by value scheme? Since class uses deep copy semantics, if number of characters in the String object is not too small, significant time is spent in copying characters and in deleting the dynamically allocated memory. This also implies that object creation and destruction is expensive. The intention is that wherever character strings are needed,String object should be used. But if cost of creation, copying, assignment and destruction of these String objects is prohibitive, clients will refrain from using the class.

To optimize implementation,We could change implementation such that when a copy of a String object is made, both the copies share the characters in the original string without actually copying them. But when one of the copies try to modify the contents then it should get separate copy of characters without affecting other objects. This scheme of making a true copy of object when modification is attempted is called Copy-On-Write(COW) or Lazy Copy.

Concept of lazy copy or copy on write is based on reference counting mechanism. Reference counting is a technique that allows multiple objects with the same value to share a single representation of that value.

```
class String
{
private:
	struct StringValue
	{
		unsigned referenceCount;
		unsigned length;
		char* buffer;
	public:
		StringValue(void)
		{
			this->referenceCount = 1;
			this->length = 0;
			this->buffer = NULL;
		}
		StringValue(const char* string)throw(bad_alloc)
		{
			this->referenceCount = 1;
			this->length = strlen(string);
			this->buffer = new char[ this->length + 1 ];
			strcpy(this->buffer, string);
		}
		~StringValue(void)
		{
			delete[] this->buffer;
			this->buffer = NULL;
		}
	};
private:
	StringValue* string;
public:
	String(void)
	{
		this->string = new StringValue();
	}
```

```
String( const char* str )
{
        this->string = new StringValue(str);
}
String(const String& other)//copy ctor
{
        //First increment the reference count
other.string->referenceCount=other.string->referenceCount + 1;
        //then share the resource
        this->string = other.string;
}
String& operator=(const String& other)
{
        if (this == &other)
                return *this;
        this->~String();
other.string->referenceCount=other.string->referenceCount + 1;
        this->string = other.string;
        return *this;
}
String& toLower(void)      //Copy-On-Write Strategy
{
        if (this->string->_referenceCount > 1)
        {
                char* str = this->string->_buffer;
                this->string->_referenceCount = this->string->_referenceCount - 1;
                this->string = new SStringValue(str);
        }
        char* str = this->string->_buffer;
        while (*str != '\0')
        {
                *str = tolower(*str);
                ++str;
        }
        *str = '\0';
        return *this;
```

```
        }
        ~String(void)
        {
this->string->referenceCount = this->string->referenceCount - 1;
                if (this->string->referenceCount == 0)
                        delete this->string;
                this->string = NULL;
        }
};
int main(void)
{
        String str1("Sandeep Kulange");
        str1 = str1.toLower();
        return 0;}
```

Test Your Understanding:

1. Point pt1;
   Point* ptr = &pt1;

2. Point pt1;
   Point& pt2 = pt1;

3. Point pt1;
   Point pt2(pt1);

4. Point pt1;
   Point pt2 = pt1;

5. Point pt1,pt2;
   pt2 = pt1;

Static Members

"Scope" decides the area or region in which we can access the portion of the program. Following are the types of scope:

1.Statement Scope

2.Block Scope

3.Function Scope
4.Class Scope
5.Namespace Scope
6.File Scope

"Lifetime" is the period during execution of a program in which a variable or function exists. Storage classes govern the scope and lifetime of an object in C as well as C++.Following are the storage classes in C/C++.

1.auto
2.static
3.register
4.extern

Local variables are also called automatic variables becoz if we give call to the function then its memory gets managed automatically. Consider the following example:

```
void print(void)
{
        auto count = 0;
        ++count;
        cout << "Count     :       " << count << endl;
}
int main(void)
{
        ::print();      //1
        ::print();      //1
        ::print();      //1
        return 0;
}
```

If we want to persist value of the variable during function call then we should use static storage class. static variables are same as global variable but it is having limited scope Consider the following example:

```
void print(void)
{
        static int count = 0;
        ++count;
        cout << "Count     :       " << count << endl;
```

```
}
int main(void)
{
        ::print();      //1
        ::print();      //2
        ::print();      //3
        return 0;
}
```

We can declare global function as a static. In this case we cannot access it outside the file.
We can declare main function as static but compiler ignores it.

Non static data member is called instance variable and non static member function is called instance method. To access instance members(instance variable & method), inside member function we should use this pointer and inside non member function we should use object/instance of a class.

static data member is called class level variable and static member function is also called class level method. To access class level members(class level variable & method), inside member function as well as non member function we should use class name and scope resolution operator.

Static Data Member:

If we declare non static data member inside the class then every data member gets space inside object.

If we want to share value of any data member among all the objects of same type then we should declare data member as static. In other words all the objects of same type shares single copy of static data member. To declare data member as a static it is compulsory provide global definition for it. Because all the static variable/object gets space before starting execution of main.

Consider the following example:

```
namespace NTest
{
        class Test
        {
        private:
                static int count;
        };
```

```
        int Test::count = 0;//global definition
}
```

If we forget to provide global definition then linker produces "unresovled external symbol" error.

Note:Generally we should declare constant data member as static.

```
namespace NMath
{
        class Math
        {
        private:
                static const float PI;
        };
        const float Math::PI = 3.142f;//global definition
}
```

In c++, only static constant objects can be initialized at the time of declaration.

```
namespace NMath
{
        class Math
        {
        private:
                static const int count = 10;
                //Here type must be static const
        };
}
```

Static Member Function:

To access non static data members we should define non static methods inside the class and to access static data members outside the class we should declare static method inside the class. Since static methods are designed to call using class name only, it do not get this pointer. Consider the following code:

```
namespace NInstanceCount
{
        class InstanceCount
        {
        private:
                static int instanceCount;
```

```
        public:
            instanceCount(void)
            {
InstanceCount::instanceCount = InstanceCount::instanceCount + 1;
            }
            static int getInstanceCount(void)
            {
                return InstanceCount::instanceCount;
            }
        };
        int InstanceCount::instanceCount = 0; //global definition
    }
    Int main(void)
    {
        using namespace NInstanceCount;
        cout << "Instance Count :      " << InstanceCount::getInstanceCount()   <<
endl;  //0
        InstanceCount obj1, obj2, obj3;
        cout << "Instance Count :      " << InstanceCount::getInstanceCount()   <<
endl;  //3
        return 0;
    }
Inside non static member function we can access both static as well as non static members.
    namespace NTest
    {
        class Test
        {
        private:
            int num1;
            static int num2;
        public:
            Test(void)
            {
                this->num1 = 10;
            Test::num2 = 20;  //Now old value of num2 will be over written with 20.
            }
```

```
                void Print(void)
                {
                        cout << "Num1    :       " << this->num1 << endl;
                        cout << "Num2    :       " << Test::num2 << endl;       //Allowed
                }
        };
        int Test::num2;     //First it will be initialized to 0
}
```

Since static member function do not get this pointer we cannot access non static members inside static member function directly. To access non static members inside static member function we should create object of the class. Consider the following code.

```
namespace NTest
{
        class Test
        {
        private:
                int num1;
                int num2;
        private:
                Test(void)
                {
                        this->num1 = 10;
                        this->num2 = 20;
                }
        public:
                void print(void)
                {
                        cout << "Num1    :       " << this->num1 << endl;
                        cout << "Num2    :       " << this->num2 << endl;
                }
                static void invoke()
                {
                        Test test;      //Instantiation
                        test.print();  //Now it is allowed
                }
        };
```

```
}
int main(void)
{
        using namespace NTest;
        Test::invoke();
        return 0;
}
```

We cannot declare static member function as constant, volatile or virtual.

If we don't want to modify state of the current object inside member function then generally we declare member function as constant. Since static member functions are not designed to call using object we cannot declare static member function as constant.

Even though static members are designed to call using class name, it is possible to access static members using object name.

Note:Generally we should use classname and scope resolution operator to access static members.

Design Pattern And Its Classification:

Design patterns are not algorithms or components. The design patterns are language independent strategies for solving common object oriented design problems. Here is the classification of design patterns which is listed in book of GOF.

1. Creational Design Patterns
    i.Abstract Factory
    ii.Builder
    iii.Factory Method
    iv.Prototype
    v.Singleton
2. Structural Design Patterns
    i.Adapter
    ii.Bridge
    iii.Composite
    iv.Decorator
    v.Facade
    vi.Flyweight
    vii.Proxy
3. Behavioural Design Patterns
    I.Chain of responsiblity
    ii.Command

iii.Interpreter
iv.Iterator
v.Mediator
vi.Memento
vii.Observer
viii.State
ix.Strategy
X.Template method
xi.Visitor

Singleton class: A class which allows us to create only one instance of it and which provides global point of access to it is called singleton class.

```
namespace NSingleton
{
	class Singleton
	{
	private:
		static Singleton* ptrSingletonInstance;
	private:
		Singleton()
		{	}
	public:
		static Singleton* getInstance()
		{
			if (Singleton::ptrSingletonInstance == NULL)
				Singleton::ptrSingletonInstance = new Singleton();
			return Singleton::ptrSingletonInstance;
		}
	};
	Singleton* Singleton::ptrSingletonInstance = NULL;
}
```

Test Your Understanding:

1.Why static member function do not get this pointer?

2.Why we cannot declare static member function as a constant.

3.Observe the following code and decide wheather it is overloading or not?

```
class Test
{
public:
        void print();
        static void print();
};
```

Friend function and class:

If we want to access private members of the class inside non member function then we can use any one of the following:

1. Inspector/mutator function
2. Friend function/class
3. Pointer.

Already we have discussed how to use inspector and mutator function. We will discuss pointer part later.

Friend function:

A non member function of a class which is designed to access private and protected members of the class is called friend function.friend is a keyword in C++.

1. Including main function we can declare any global function as friend. But generally we should not declare main function as a friend. Consider the following code:

```
namespace NTest
{
        class Test
        {
        private:
                int num1;
                int num2;
        public:
                Test(void) : num1(10), num2(20)
                {       }
                friend void sum(void);
        };
        void sum(void)
        {
```

```
            Test test;
            int result = test.num1 + test.num2;
            cout << "Result     :        " << result << endl;
        }
    }
    int main(void)
    {
        NTest::sum();
        return 0; }
```

Friend declaration statement(friend void sum(void);)may appear in any region of the class(private, protected, public).
If we want to declare any function as a friend then class and function must be exist inside same namespace.

2. Member function of a class can be declared as friend function inside another class.
Note:It is trick. People generally preferred friend class.

```
class A
{
public:
    void sum(void);
    void sub(void);
};
class B
{
private:
    int num1;
    int num2;
public:
    B();
    friend void A::sum(void);
    friend void A::sub(void);
};
```

3. A Class can be declared as friend class of another class.

```
class A
```

```
{
public:
        void sum(void);
        void sub(void);
};
class B
{
private:
        int num1;
        int num2;
public:
        B();
        friend class A;
};
```

4. We cannot declare mutual friend functions but we can declare mutual friend classes.

```
class B;        //forward declaration
class A
{
private:
        int number;
public:
        void print();
        friend class B;
};
class B
{
private:
        int number;
public:
        void print();
        friend class A;
};
```

To access private or protected elements of the class inside another non member function or class, we declare function or class as a friend. With the help of friend function we cannot access members of function so we cannot declare mutual friend functions.

we can declare single function as a friend inside multiple classes.

```
class Complex
{
private:
        int real;
        int imag;
public:
        Complex(void) : real(10), imag(20)
        {       }
        friend void sum();
};
class Point
{
private:
        int xPosition;
        int yPosition;
public:
        Point(void) : xPosition(30), yPosition(40)
        {       }
        friend void sum();
};
void sum()
{
        //TODO
}
```

Linked list is a collection of items/elements where each item/element is called node. Node is an object which may consist of either two parts or three parts depending on type of linked list.

A pointer which stores address of first node is called head pointer. If value of head is NULL then linked list is considered as empty.

A pointer which stores address of next node is called tail pointer.

There are two types of linked list:

1. Singly linked list.
2. Doubly linked list.

Singly Linked List : In a linked list if node consist of two parts i.e.
1. Data
2. A pointer which stores address of next node(next)

Then such type of linked list is called singly linked list.

Doubly Linked List : In a linked list if node consist of three parts i.e.
1.A pointer which stores address of previous node(prev)
2.Data
3.A pointer which stores address of next node(next) then such type of linked list is called doubly linked list.

In a singly linked list, if next pointer of last node stores NULL value then it is called linear singly linked list.

In a singly linked list, if next pointer of last node stores address of first node then it is called circular singly linked list.

In a doubly linked list, if previous pointer of first node and next pointer of last node stores NULL value then it is called linear doubly linked list.

In a doubly linked list, if previous pointer of first node stores address of last node and next pointer of last node stores address of first node then it is called circular doubly linked list.

Process of visiting nodes in linked list is called traversing.Generally to traverse linked list or any collection we should use iterator. We will discuss iterator in operator overloading.

```
namespace NList
{
        class List;     //Forward declaration

class Node
        {
        private:
                int data;
                Node* next;
        public:
                Node(const int data = 0) : data(data), next(NULL)
                {       }
```

```
        friend class List;
    };
    class List
    {
    private:
        Node* head;
        Node* tail;
    public:
        List(void) :head(NULL), tail(NULL)
        {       }
        bool empty(void)const
        {
            return this->head == NULL;
        }
        void addFirst(const int data)throw( bad_alloc)
        {
            Node* newnode = new Node(data);
            if (this->empty())
                this->tail = newnode;
            else
                newnode->next = this->head;
            this->head = newnode;
        }
    };
}
int main(void)
{
    using namespace NList;
    List list;
    list.addFirst(10);
    list.addFirst(20);
    return 0;
}
```

Test your understanding:
Why friend function do not get this pointer?

Template:

C++ requires us to declare variables, functions, and most other kinds of entities using specific types. However, a lot of code looks the same for different types. Especially if we implement algorithms, such as quicksort, or if we implement a linked list or a binary tree for different types, the code looks the same despite the type used.

If language doesn't support a special feature for this, we only have bad alternatives:

1. we can implement the same behavior again and again for each type.
2. we can write general code using void*.
3. we can use special preprocessors.

Consider the following example:

```
void swap_object(int& object1, int& object2)
{
        int temp = object1;
        object1 = object2;
        object2 = temp;
}
void swap_object(string& object1, string& object2)
{
        string temp = object1;
        object1 = object2;
        object2 = temp;
}
```

In this case we can save programmers effort with the help of template. When we use a template, we can pass types as arguments. In other words "Parameterized type is also called template". we can write above code with the help of template.

```
template< typename Type >
void swap_object(Type& object1, Type& object2)
{
        Type temp = object1;
        object1 = object2;
        object2 = temp;
}
```

Now it is not a specific function it is generic function. More specific it is a Function template.
As shown in above code , template parameters must be announced with syntax of the following form:

template < comma-separated-list-of-parameters >

In our example, the list of parameters is typename Type. The keyword typename introduces a so-called type parameter. We can use any identifier as a parameter name, but using T is the convention.

```
int main( void )
{
	int num1 = 10;
	int num2 = 20;
	swap_object<int>( num1, num2 );	//Template argument list
	return 0;
}
```

At compile time Type will be replaced with int. Here swap_object<int>(...); indicated template argument list.
For historical reasons, you can also use class instead of typename to define a type parameter. The keyword
typename came relatively late in the evolution of the C++ language.
With the help of template we cannot save space or time rather we can save programmers effort only.
Not only function but also class can can be declared as template class.

```
template< class Type >
class Stack
{
private:
	int top;
	int size;
	Type* arr;
public:
	Stack( void );
	bool empty( void ) const;
```

```
        bool full( void ) const;
        void push( Type data );
        Type peek( void );
        void pop( void );
};
```

For class templates we can also define default values for template parameters. These values are called default template arguments.

```
template< class Type = int>
class Stack
{

};
int main(void)
{
        Stack<> s1;
        return 0;
}
```

Note:We cannot divide implementation of template class into multiple files. declaration and definition must be in same file.

Operator Overloading:

In C as well as C++ we can use operators with the objects of fundamental types only. In C we cannot use operators with the objects of user defined type directly(except sizeof).
In C++ if we want to use operators with objects then it is necessary to overload an operator. And to overload an operator it is necessary to define operator function. Now, we can define operator function as a member or non member function.

if we write pt3 = pt1 + pt2 then compiler can assume it like

```
        pt3 = pt1.operator+( pt2 );
        pt3 = operator+( pt1, pt2 );
```

operator is keyword in C++. If we want use + operator with objects then we should define logic of addition inside operator+() function. Now where to write definition of operator function is upto the programmer.

Consider the following code.

```
pt3 = pt1 + pt2;            //pt3 = pt1.operator+( pt2 );
```

in this case operator function will look like as follows

```
class Point
{
private:
        int xPosition;
        int yPosition;
public:
        //Point* const this = &pt1;
        //const Point& pointInstance = pt2;
        Point operator+( const CPoint& pointInstance )const
        {
                Point temp;
                temp.xPosition = this->xPosition + pointInstance.xPosition;
                temp.yPosition = this->yPosition + pointInstance.yPosition;
                return temp;
        }
};
```

And if we write

```
pt3 = pt1 + pt2;            //pt3 = operator+( pt1, pt2 );
```

in this case operator function will look like as follows

```
class Point
{
private:
        int xPosition;
        int yPosition;
public:
        //const Point& pt1 = pt1;
        //const Point& pt2 = pt2;
        friend Point operator+(const Point& pt1, const Point& pt2)
        {
                Point temp;
                temp.xPosition = pt1.xPosition + pt2.xPosition;
                temp.yPosition = pt1.yPosition + pt2.yPosition;
```

```
        return temp;
    }
};
```

With the help of operator overloading we cannot create new operator rather we can increase capability of the existing operators.

"Process of giving extension to the meaning of the operator is called operator overloading".

Limitations Of Operator Overloading:

1.We cannot overload following operators as a member or non member function.

a. Member selection operator. (.)
b. Pointer to member selection operator. (.*)
c. Conditional/Ternary operator. (? :)
d. Scope resolution operator. (::)
e. sizeof operator.
f. typeid operator.
g. static_cast operator.
h. dynamic_cast operator.
i. const_cast operator.
j. reinterpret_cast operator.

2.We cannot overload following operators as a non member function.

a. Assignment operator. (=)
b. Index/Subscript operator. ([])
c. Function call operator. (())
d. Dereferencing operator. (->)

3.We can change meaning of the operator with the help of operator overloading.

First we will discuss overloading by implementing operator function as a member of class.

If we want to overload binary operator using member function then operator function should take only one parameter.

If we want to overload unary operator using member function then operator function should not take any paramater.

Overloading Arithmetic Operator:

```
1.  pt3 = pt1 + pt2;          //pt3 = pt1.operator+( pt2 );
2.  pt4 = pt1 + pt2 + pt3;          //pt4 = pt1.operator+( pt2 ).operator+( pt3 );
3.  pt3 = pt1 - pt2;          //pt3 = pt1.operator-( pt2 );
4.  pt3 = pt1 * pt2;          //pt3 = pt1.operator*( pt2 );
5.  pt3 = pt1 / pt2;          //pt3 = pt1.operator/( pt2 );
6.  pt3 = pt1 % pt2;          //pt3 = pt1.operator%( pt2 );
7.  pt2 = pt1 + 5;                  //pt2 = pt1.operator+( 5 );
8.  pt2 = 5 + pt1;                  //Not allowed;
```

Overloading Relational Operator:

```
1.  bool result = pt1 == pt2;       //bool result = pt1.operator==( pt2 );
2.  bool result = pt1 != pt2;       //bool result = pt1.operator!=( pt2 );
3.  bool result = pt1 > pt2;        //bool result = pt1.operator>( pt2 );
4.  bool result = pt1 < pt2;        //bool result = pt1.operator<( pt2 );
5.  bool result = pt1 >= pt2;       //bool result = pt1.operator>=( pt2 );
6.  bool result = pt1 <= pt2;       //bool result = pt1.operator<=( pt2 );
```

Overloading Short Hand Operators:

```
1.  pt1 += pt2;               //pt1.operator+=( pt2 );
2.  pt1 -= pt2;               //pt1.operator-=( pt2 );
3.  pt1 *= pt2;               //pt1.operator*=( pt2 );
```

Overloading Unary Operators:

```
1.  pt2 = ++ pt1;                   //pt2 = pt1.operator++( );
2.  pt2 = -- pt1;             //pt2 = pt1.operator--( );
3.  pt2 = pt1 ++;                   //pt2 = pt1.operator++( 0 );
4.  pt2 = pt1 --;             //pt2 = pt1.operator--( 0 );
```

Now, we will discuss overloading by implementing operator function as a non member of class.

If we want to overload binary operator using non member function then operator function should take two parameters.

If we want to overload unary operator using non member function then operator function should take one parameter.

Overloading arithmetic operator:

```
1.  pt3 = pt1 + pt2;        //pt3 = operator+( pt1, pt2 );
2.  pt4 = pt1 + pt2 + pt3;      //pt4 = operator+( operator+( pt1, pt2 ),pt3 );
3.  pt3 = pt1 - pt2;        //pt3 = operator-( pt1, pt2 );
4.  pt2 = pt1 + 5;          //pt2 = operator+( pt1, 5 );
5.  pt2 = 5 + pt1;          //pt2 = operator+( 5, pt1 );
```

Overloading Relational Operator:

```
1.  bool result = pt1 == pt2;       //bool result = operator==( pt1, pt2 );
```

Overloading Short Hand Operators:

```
1.  pt1 += pt2;             //operator+=( pt1, pt2 );
```

Overloading Unary Operators:

```
1.  pt2 = ++ pt1;           //pt2 = operator++( pt1 );
2.  pt2 = -- pt1;           //pt2 = operator--( pt1, 0 );
```

There are two rules of thumb when implementing overloaded operators

1. Any operator that does not require an l-value is better implemented as non member function.
   a.Arithmetic operators
   b.Relational operators
2. Any operator that requires an l-value is better implemented as member function.
   a.Shorthand operators
   b.Increment and decrement operators
   c.Assignment,Index, Function call and dereferencing operator.

Overloading Insertion Operator (<<):

if we want to print state of an object on console then we should overload insertion operator.

```
cout<<pt1;          //operator<<( cout, pt1 );
cout<<pt1<<endl; //operator<<( cout, pt1 )<<endl;
cout<<pt1<<pt2;  //operator<<( operator<<( cout, pt1 ), pt2 )
```

If we pass object of ostream class by value then its copy constructor will call.But copy constructor of ostream class is private so it is necessary to pass and return object of ostream class by reference.

Overloading Extraction Operator(>>):

If we want to read state for an object from console then we should overload extraction operator.

```
cin>>pt1;          //operator>>( cin, pt1 );
cin>>pt1, pt2;          //operator>>( operator>>( cin, pt1 ), pt2 );
```

If we pass object of istream class by value then its copy constructor will call. But copy constructor of istream class is private so it is necessary to pass and return object of istream class by reference.

Overloading Assignment Operator:

If we initialize newly created object from existing object then copy constructor gets call.

```
Point pt2 = pt1;     //Here for pt2 object copy constructor will call.
```

If we assign existing object to another existing object then assignment operator gets call.

```
pt2 = pt1;          //Here assignment operator function gets call.
```

If we do not provide assignment operator function for the class then compiler provides one assignment operator function for that class by default. Such assignment operator function is called default assignment operator function.

Class contains following function by default:

1.default constructor
2.default destructor
3.default copy constructor
4.default assignment operator function.

Consider the assignment statement statement and its implicit call.

```
pt2 = pt1;          //pt2.operator=( pt1 );

pt3 = pt2 = pt1;    //pt3.operator=( pt2.operator=( pt1 ) );
```

We cannot overload assignment operator as non member function. Consider the following code:

```
class Point
{
private:
        ...
public:
```

```
        Point& operator=(const Point& other)
        {
                if (this == &other)
                        return *this;
                else
                {
                        this->xPosition = other.xPosition;
                        this->yPosition = other.yPosition;
                }
                return *this;
        }
};
```

In above program we have overloaded assignment operator and we have done shallow copy. If we want to do shallow copy then no need to overload assignment operator because default assignment operator do shallow copy. In case of assignment if we want to do deep copy or lazy copy then we should overload assignment operator.

Consider the following code:

```
class String
{
private:
        int length;
        char* buffer;
public:
        String& operator=(const String& other)
        {
                if (this == &other)
                        return *this;
                else
                {
                        this->~String();
                        this->length = other.length;
                        this->buffer = new char[this->length + 1];
                        strcpy(this->buffer, other.buffer);
                }
                return *this;
        }
```

```
};
```

Overloading Index/Subscript Operator:

We know that there are some disadvantages of language defined array:

1.We cannot decide size for an array at runtime.

2.The built in array type does not have any kind of subscript range checking.

3.We cannot copy arrays directly inside another array.

To avoid these limitations lets put array inside object and perform operations on object as array. Consider the following code.

```
class Array
{
private:
        int size;
        int* arr;
public:
        Array(const int size)
        {
                this->size = size;
                this->arr = new int[this->size];
        }
        int& operator[](const int index)
        {
                if (index >= 0 && index < this->_size)
                        return this->arr[index];
                throw Exception("Array index out of bounds");
        }
};
```

Now we can specify size for an array at runtime.

```
        int size = 5;
        Array a1( size );      //valid;
```

Note: When operator [] is on the left hand side of =, It must return an address(l-value) and when it is on the right side of =, it must return a value (r-value).

```
        int number = a1[ 2 ];              //int number = a1.operator[]( 2 );
```

In this case reference is act as value;

```
a1[ 2 ] = 300;                    //a1.operator[]( 2 ) = 300;
```

In this case reference is act as address.

In short, if we want to treat any object as a array then we should overload subscript or index operator. To overload index/subscript operator class should contain object of collection as a data member.

Overloading Function Call Operator:

If we treat any function as a object then it is called function object. To create function object it is necessary to overload function call operator.

```
class String
{
private:
        int length;
        char* buffer;
public:
        void operator()( const char* str )
        {
                this->~String();
                this->length = strlen(str);
                this->buffer = new char[this->length + 1];
                strcpy(this->buffer, str);
        }
};
int main(void)
{
        String str;
        str("Sandeep");      //here str is function object.
        return 0;
}
```

Overloading Dereferencing Operator:

if we want to use dot /member selection operator to access members of the class then left side oprand must be object.

e.g. pt1.print();

if we want to use arrow operator to access members of the class then left side oprand must be pointer of class.

e.g. ptr->print();

To avoid memory leakage we can use auto_ptr smart pointer. Lets write own version of auto_ptr class.

```
template< class Type >
class AutomaticPointer
{
private:
	Type* ptr;
public:
	AutomaticPointer(Type* ptr)
	{
		this->ptr = ptr;
	}
	Type* operator->(void)
	{
		return this->ptr;
	}
	~AutomaticPointer(void)
	{
		delete ptr;
	}
};
int main(void)
{
	AutomaticPointer<Complex> autoPtr(new Complex());
	ptr->print();
	return 0;
}
```

In above code autoPtr is object which act as a pointer, so it is called smart pointer.

Nestsed class :

If we define class inside scope of another class then it is called nested class.

```
class List
{
private:
	class Node   //Nested class.
```

```
        {
            ....
        };
    private:
        Node* head;
    };
```

Local class: If we define class inside scope of another function then it is called local class.

```
    void f1( void )
    {
        class SomeClass     //Local class.
        {
            ....
        };
    }
```

Iterator is a smart pointer which is used to traverse the collection. It is a behavioral design pattern.

```
    class List
    {
    private:
        class Iterator;         //Forward declaration
        class Node
        {
        private:
            int data;
            Node* next;
        public:
            friend class List;
            friend class Iterator;
        };
    public:
        class Iterator
        {
        private:
            Node* ptrNode;
        public:
            Iterator(Node* ptrNode)
```

```
		{
			this->ptrNode = ptrNode;
		}
		bool operator!=(Iterator& other)
		{
			return this->ptrNode != other.ptrNode;
		}
		int operator*(void)
		{
			return this->ptrNode->data;
		}
		void operator++(void)
		{
			this->ptrNode = this->ptrNode->next;
		}
	};
private:
	Node* head;
	Node* tail;
public:
	Iterator begin(void)
	{
		return Iterator(this->_head);
	}
	Iterator end(void)
	{
		return Iterator(NULL);
	}
};
int main(void)
{
	List list;
	List::Iterator itrStart = list.begin();
	List::Iterator itrEnd = list.end();
	while (itrStart != itrEnd)
	{
```

```
            cout << *itrStart << endl;      //itrStart : smart pointer
            ++itrStart;
        }
        return 0;
    }
```

Conversion Function:

A member function of the class which is used to convert state of an object one type into another. In C++, there are three conversion functions:

1.Single parameter constructor.

```
int num1 = 10;
Point pt1 = num1;
```

In above statement, single parameter constructor is responsible for converting value of num1 into pt1 so single parameter constructor is also called conversion constructor.

2. Assignment Operator:

```
int num1 = 10;
Point pt1;
pt1 = num1; //pt1.operator=( Point( num1 ) );
```

In this case assignment operator is responsible for converting state of num1 into pt1 so it is called conversion function. To convert state of num1 into pt1, assignment takes help of anonymous object. If we want to keep control on anonymous object then we should declare single parameter constructor as explicit.

3. Type Conversion operator.

```
Point pt1( 10,20 );
int xPosition = pt1;//      int xPosition = pt1.operator int();
```

In this case type conversion operator (operator int( )) is responsible for converting state of pt1 into xPosition so it is called conversion function.

C++ - Hierarchy Notes

If "**has-a**" relationship exists between two types then we should declare object as data member.

e.g. Employee has a joinDate.

```
class Date
{
private:
    int day, month, year;
};//End of Date class
class Employee
{
private:
    string name;          //Composition
    int empid;
    float salary;
    Date joinDate;        //Composition
};//End of class Employee
```

If "**is-a**" relationship exists between two types then we should use inheritance. e.g Employee is a Person.

```
class Person                        //Parent class
{
private:
    string name;
    int age;
public:
    //TODO
};
class Employee : public Person  //Child class
{
private:
    int empid;
    float salary;
public:
    //TODO
};
```

Is-a hierarchy is also called as "**kind-of** " hierarchy.

"Journey from generalization to specialization is called as inheritance".

C++ - Hierarchy Notes

Inheritance is one of the most powerful concepts in object oriented software development. If the concept is clearly understood then it can be used very effectively in number of design scenarios.

```
class Person                          //Parent / Base class
{
private:
    string name;
    int age;
public:
    Person( void ) : name(""),age(0)
    {   }
    Person( string name, int age ) : name(name),age(age)
    {   }
    void printRecord( void )
    {
        cout<<"Name :    "<<this->name<<endl;
        cout<<"Age  :    "<<this->age<<endl;
    }
};
```

Inheritance is one of the fundamental tool for code reusability. Lets consider the simple code for inheritance.

```
class Employee : public Person  //Derived / Child class
{
private:
    int empid;
    float salary;
public:
    Employee( void ) : empid( 0 ), salary( 0.0f )
    {   }
    Employee( string name, int age, int empid, float salary ) : Person( name, age ),//Ctor Base init list
            empid( empid ), salary( salary )    //Ctor member initializer list
    {   }
    void printRecord( void )
    {
        Person::printRecord();  //Call to Base class function
        cout<<"Empid    :   "<<this->empid<<endl;
        cout<<"Salary   :   "<<this->salary<<endl;
    }
};
```

In C++, parent class is called as Base class and Child class is called as Derived class. According to this concept, class Person is Base class and class Employee class is Derived class.

In inheritance, all static and non static data memebrs of base class inherits into the derived class.

If data member in base class is static then we can access it using base class name as well as derived classname but it do not get space inside object. Consider the following code:

```
class Base
{
protected:
    static int number;
public:
    void showRecord( void )
    {
        cout<<"Num1 :   "<<Base::number<<endl;
    }
};
int Base::number = 10;  //Global definition
class Derived : public Base
{
public:
    void displayRecord( void )
    {
        cout<<"Num1 :   "<<Derived::number<<endl;
    }
};
```

If data member in base class is non static then it inherits into the derived class and it gets space in derived class object.

Size of derived class object = size of all non static data members declared in base class + size of all non static data members declared in derived class.

Data members of base class inherits into the derived class hence it affects on the size of object but members of derived class do not inherit into the base class hence it do not affects on size of object.

"If we use inheritance then data members inherits into the derived class."

If we want to access data members of base in derived class and If name of base class data member and derived class data member is same then we should use class name and scope resolution operator

If name of base class data members and derived class data mebers are different then to access data members of base we can use either class name or this pointer. In general to access members of base class we should use class name. Consider the following code:

```
class Base
{
protected:
    int num1;
};
class Derived : public Base
{
private:
    int num1;
public:
    Derived( void )
    {
        Base::num1 = 10;
        this->num1 = 20;
    }
    void printRecord( void )
    {
        cout<<"Num1 :    "<<Base::num1<<endl;
        cout<<"Num1 :    "<<this->num1<<endl;
    }
};
```

Now lets consider the member function. Except few function all static and non static member function of base class inherits itnto the derived class.

Following function do not inherit into the derived class:

1. Constructor
2. Destructor
3. Copy constructor
4. Assignnment operator
5. Friend function

If we declare static member function in base class and if we want to access it in non member function then we can access it using base class as well as derived class name and scope resolution operator.

In general static data member and member function of base class inherits into the derived class.

Non static member function of base class inherits into the derived class hence using derived class object we can access non static member function of base class as well as derived class.

In general non static data member and member function of base class inherits into the derived class.

Consider the following code:

```
class Base
{
public:
    void showRecord( void )
    {   }
};
class Derived : public Base
{
public:
    void displayRecord( void )
    {   }
};
int main()
{
    Derived derived;

    derived.showRecord();   //Base::showRecord

    derived.displayRecord();    //Derived::displayRecord
    return 0;
}
```

If name of base class member function and derived class member function is same then derived class member function hides the implementation of base class member function which is inherited in derived class hence priority is always given to derived class member function. ***It is called as method hiding or shadowing.***

In this case, to access member function of base inside derived class, we should class name and scope resolution operator.

"In general, to access members of base class inside member function of derived class, we should use class name and scope resolution operator."

Applications of scope resolution operator[::]

1. To access members of namespace
2. To define member function globaly
3. To access static members
4. To access members of base class in derived class.

Consider the following code:

```
class Base
{
public:
    void printRecord( void )
    {   }
};
class Derived : public Base
{
public:
    void printRecord( void )
    {
        Base::printRecord();    //call to base class member function
        //TODO : print members of derived class.
    }
};
int main()
{
    Derived derived;

    derived.printRecord();
    return 0;
}
```

if we want to access printRecord member function of base class and derived class in non member function then we should use following syntax.

```
int main()
{
    Derived derived;

    derived.Base::printRecord();     //Call to Base class function

    derived.Derived::printRecord(); //Call to Derived class function

    return 0;
}
```

Already we have seen constructor calling sequence is depends on order of objects and destructor calling sequence is excatly opposite of constructor calling sequence.

```
class Base
{
private:
    int num1;
    int num2;
public:
    Base( void ) : num1( 0 ), num2( 0 ) //ctors member initializer list
    {   }
    Base( int num1, int num2 ) : num1( num1 ), num2( num2 ) //ctors member initializer list
    {   }
};
class Derived : public Base
{
private:
    int num3;
public:
    Derived( void ) : num3( 0 )//ctors member initializer list
    {   }
    Derived( int num1, int num2, int num3 ) :
        Base(num1, num2 ),//ctors base initializer list
        num3( num3 )//ctors member initializer list
    {   }
};
```

If we create object of derived class then first base class constructor gets called and then derived class constructor gets called. Destructor calling sequence is exactly opposite i.e. first derived class destructor gets called and then base class destructor gets called.

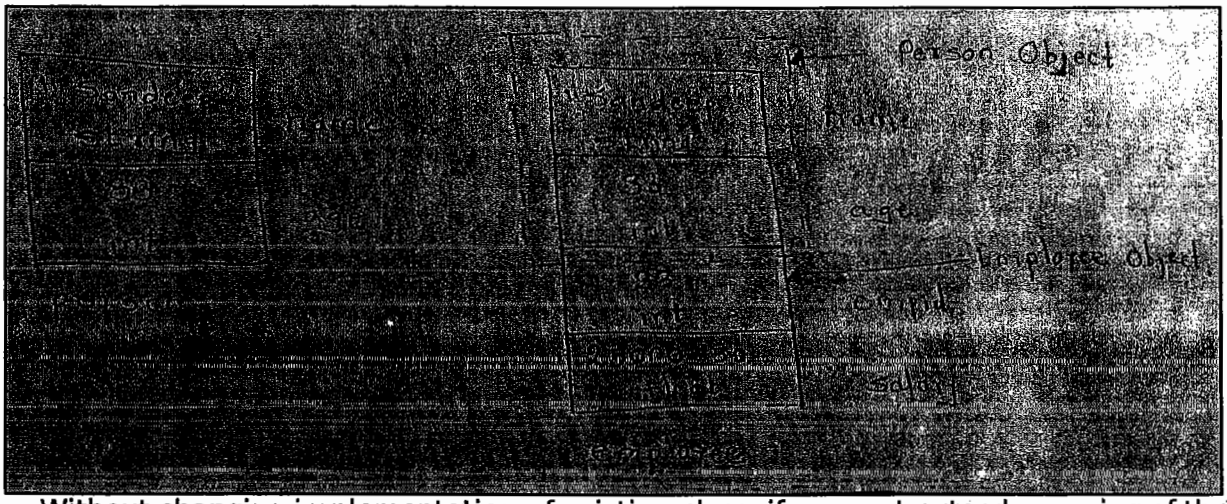
By default, from any derived class constructor(parameter less or parameterized), base class's parameter less constructor gets called. If we want to call any constructor of base class from derived class then we should use constructors base initializer list. Consider the following code:

Constructors base initializer list indicates explicit call to the constructor. As shown in above code we can use member initializer list and base initializer list together.

Constructor's member initializer list – It is used to initialize data members of same class.

Constructor's base initializer list – It is used to initialize data members of base class.

In inheritance, members of base class inherits into the derived class and hence every derived class object can be considered as base class object. e.g. Every employee is person.



Without changing implementation of existing class, if we want extend meaning of the class then we should use inheritance. In other words, if implementation of existing class is partially complete and without modifying its implementation if we want to make it complete then we should create its derived class i.e. we should use inheritance.

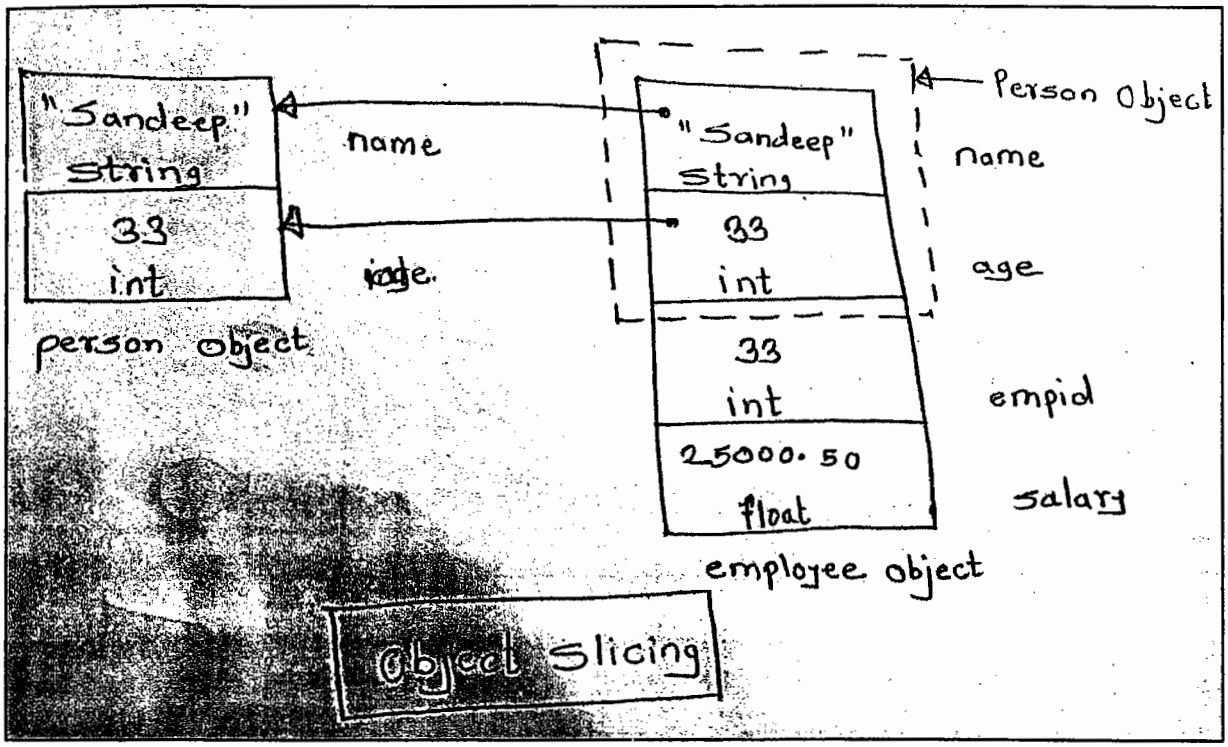
In inheritance, members of base class inherits into the derived class hence every derived class object can be considered as base class object. Since every derived class object can be treated as base class object, we can assign derived class object to the base class object.

```
Person p1("Sandeep",33);
Person p2 = p1;                    //OK

Employee emp1("Sandeep Kulange",33,33,25000.50f);
Employee emp2 = emp1;              //OK

Employee employee("Sandeep Kulange",33,33,25000.50f);
Person person = employee;          // Object Slicing
```

Object Slicing



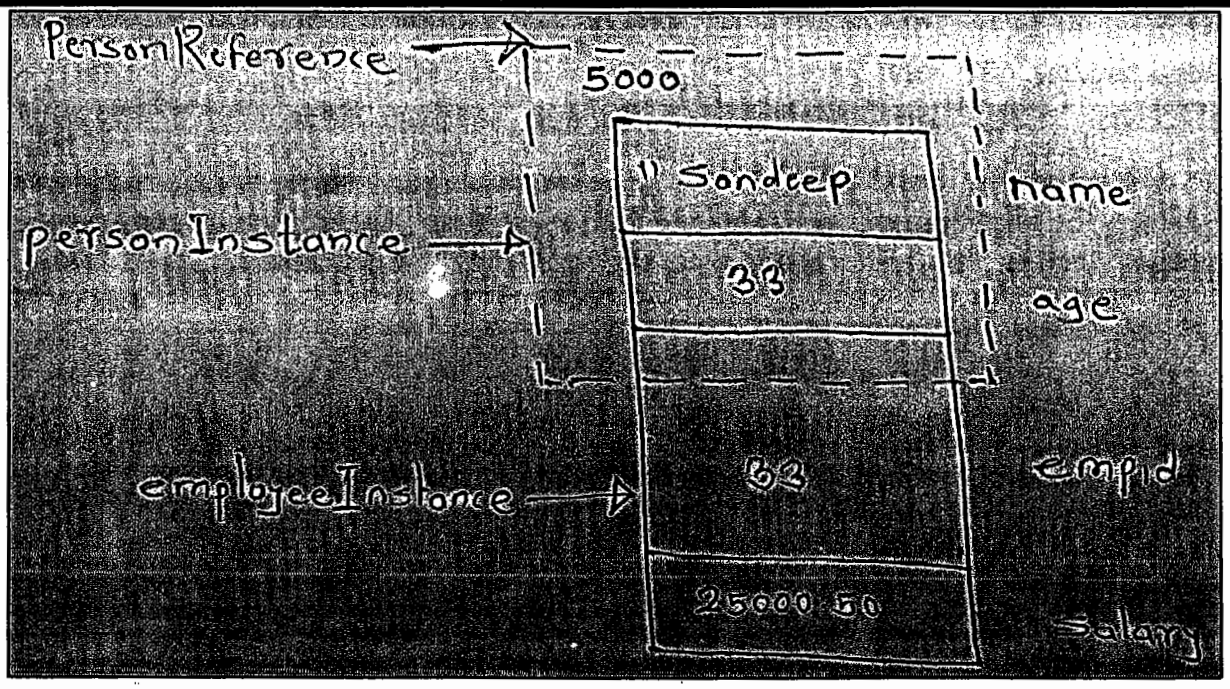
If we try to assign derived class object to the base class object, then compiler copies only values of base class members from derived object into base class object. Such process is called **object slicing**.

By declaring base class as a abstract, we can avoid object slicing.

Let us see object slicing in case of reference.

Employee employeeInstance("Sandeep",33,33,25000.50f);

Person&personReference= employeeInstance;



as shown in above code, personReference is an alias for the person instance exist inside employeeInstance. It is a form of object slicing.

Friend function do not inherit into the derived class.Lets see, how object slicing help us to overload insertion and extraction operator in inheritance.

```
class Person
{
private:
    string name;
    int age;
public:
    Person( void ) : name( " " ), age( 0 )
    {   }
    friend istream& operator>>( istream& cin, Person& other )
    {
        cout<<"Name      :       ";
        cin>>other.name;
        cout<<"Age       :       ";
        cin>>other.age;
        return cin;
    }
    friend ostream& operator<<( ostream& cout, const Person& other )
    {
        cout<<other.name<<"      "<<other.age<<" ";
        return cout;
    }
};
```

```
class Employee : public Person
{
private:
    int empid;
    float salary;
public:
    Employee( void ) : empid( 0 ), salary( 0 )
    {   }
    friend istream& operator>>( istream& cin, Employee& other )
    {
        Person& personRef = (Person&)( other );
        cin>>personRef;          //operator>>( cin, personRef );
        cout<<"Empid    :        ";
        cin>>other.empid;
        cout<<"Salary   :        ";
        cin>>other.salary;
        return cin;
    }
    friend ostream& operator<<( ostream& cout, const Employee& other )
    {
        const Person& personRef = (Person&)( other );
        cout<<personRef;         //operator<<( cout, personRef );
        cout<<other.empid<<"    "<<other.salary<<"       ";
        return cout;
    }
};
```

Consider the following code:

```
class Employee : public Person
```

We can read above statement using two ways.

1. class Person is inherited into class Employee or
2. class Employee is derived from class Person.[**Recommended**]

Direct & indirect base class and derived class :

```
class Person
{       };
class Employee : public Person
{       };
class Manager : public Employee
{       };
```

as shown in above code, class Employee is **direct** and class Person is **indirect** base class of class Manager.

Class Employee is **direct** and class Manager is **indirect** derived class of class Person.

Mode of Inheritance:

If we use private public and protected keywords to extend the class then it is called as mode of inheritance.

```
class Employee : public Person
```

Here mode of inheritance is public.

C++ supports three types of mode:

1. private mode of inheritance.[**Default mode of inheritance**]
2. protected mode of inheritance
3. public mode of Inheritance

Public mode of inheritance :if "is-a" hierarchy or relationship exists between two types then we should use public mode of inheritance. e.g. rectangle is-a shape or circle is-a shape.

```
class Shape
{       };
class Rectangle : public Shape
{       };
class Circle : public Shape
{       };
```

Public mode of inheritance				
	Same class	**Derived class**	**Indirect derived class**	**Non-member function**
Private	A	NA	NA	NA
Protected	A	A	A	NA
Public	A	A	A	A

Note : A – Accessible NA – Not Accessible

Private mode of inheritance: If "has-a" hierarchy / relationship is exist between two types then either we should use composition or private mode of inheritance.

It is default mode of inheritance in C++. Consider the following code.

```
class Date
{
    //TODO
};
class Employee : Date        //Here Default mode is private
{
    //TODO
};
```

Private mode of inheritance				
	Same class	**Derived class**	**Indirect derived class**	**Non member function**
Private	A	NA	NA	NA
Protected	A	A	NA	NA
Public	A	A	NA	A using Base Object
				NA using Derived Object

Note : *In case of has-a relationship, generally we should use composition.*

Protected mode of inheritance: It is rarely used. It is only used when you want to polymorphism to work only within the derived class but not for others. In case of public inheritance, an object of derived class can be used in place of object of base class by any client. With protected mode of inheritance, the conversion from derived to base will be allowed only within the methods of derived class.

Protected mode of inheritance				
	Same class	**Derived class**	**Indirect derived class**	**Non member function**
Private	A	NA	NA	NA
Protected	A	A	A	NA
Public	A	A	A	A using Base Object
				NA using Derived Object

Any kind of mode, we can access private members inside declared class only. If we want to access private members in derived class then either we should use member function or we should declared derived class as a friend inside base classs.

Suppose B and C are derived from class A. We want to access members of class A in class B but not in C. Consider the following code :

```
class A
{
private:
    int num1;
public:
    A( void ) : num1( 10 )
    {   }
    friend class B;
};
class B : public A
{
private:
    int num2;
public:
    B( void ) : num2( 20 )
    {   }
    void print( void )
    {
        cout<<this->num1<<"      "<<this->num2<<endl;
    }
};
```

```
class C : public A
{
private:
    int num3;
public:
    C( void ) : num3( 30 )
    {   }
    void print( void )
    {
        cout<<this->num1<<"      "<<this->num3<<endl;     //Error
    }
};
```

class B is declared as friend in class A but Class C is not declared as friend in class A. Hence members of A are accessible in class B but not in class C

Types of inheritance

Inheritance can be classified as either ***implementation inheritance*** and***interface inheritance.*** Under these catogories following are the types of inheritance.[We will discuss implementation and interface inheritance later.]

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance.

Let us discuss each type in detail.

1. Single Inheritance :

e.g.class B is derived from class A.

Syntax :

```
class A                                  //Base class
{      };
class Derived : public Base              //Derived class
{      };
```

If single base class is having single derived class then such inheritance is called single inheritance.

2. Multiple Inheritance :

e.g. class D is derived from class A,B and C.

Syntax:

```
class A{      };
class B{      };
class C{      }
class D : public A,publicB,public C
{       };
```

If multiple Base classes are having single derived class then such inheritance is called multiple inheritance.

3. Hierarchical inheritance :

e.g.class B, C and D are derived from class A.

Syntax :

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

If single base class is having multiple derived classes then such inheritance is called hierarchical inheritance.

4. Multilevel inheritance :

e.g. class B is derived from class A, class C is derived from class B and class D is derived from class C.

Syntax :

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

If single inheritance is having number of levels then it is called multilevel inheritance.

Hybrid inheritance: In C++ we can combine any two or more than two types of inheritance.

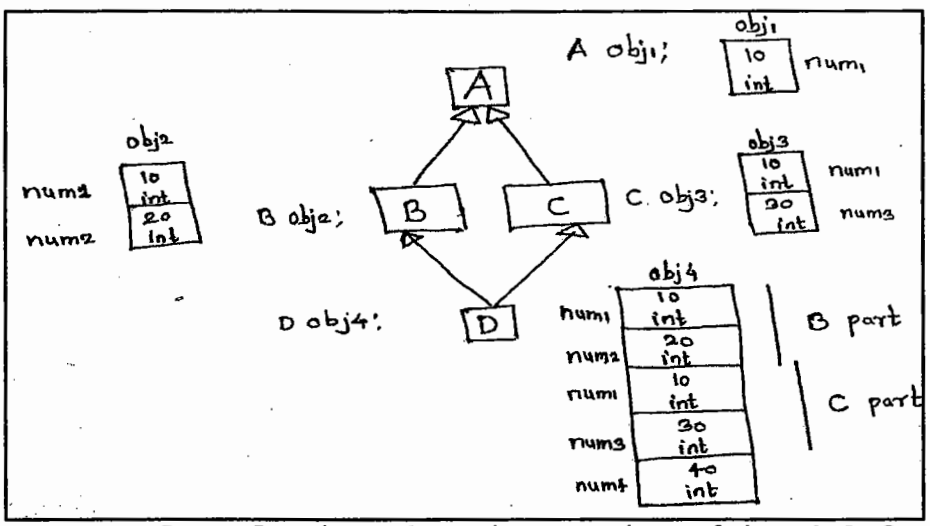
e.g. class istream&ostream are derived from class ios and class iostream is derived from class istream and ostream.

Syntax:

```
classios{      };
classistream : public ios
{      };
classostream : public ios
{      };
classiostream : public istream, public ostream
{      };
```

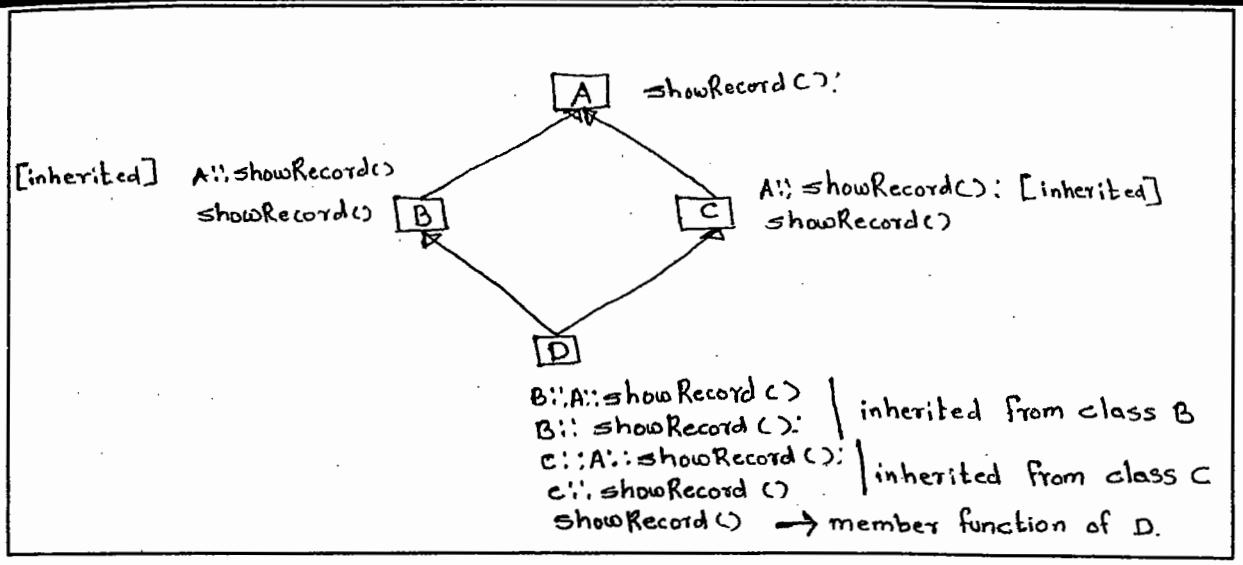
Combination of any two or more than two types of inheritance is called hybrid inheritance.

Diamond problem : suppose class B & C are derived from class A and class D is derived from class B & C. It is hybrid inheritance[Lets say it is diamond inheritance].



numz, num2,num3 and num4 are data members of class A, B, C and D respectivly.

ShowRecord() is member function declared in A,B,C as well as D class.



If we create instance of D class then constructor and destructor calling sequence is as follows:

Ctor calling sequence is : A --> B -->A -->C---> D

Dtor calling sequence is : D --> C -->A -->B---> A

i.e. constructor and destructor of class A will be call muliple times.

Lets list out the problems faced with diamond inheritace

1. datamemebrs of indirect base class inherits into the indirect derived class multiple times, hence it affects on size of object.
2. Member functions of indirect base class inherits into the indirect derived class multiple times,hence using instance of indirect derived class, if we try to access it then compiler generates ambiguity error.
3. Constructor and destructor of indirect derived class gets called multiple times.

All such problems created by hybrid inheritance is called diamond problem.

Solution of diamond problem

To overcome diamond problem we should declare base class[**class A**] as a virtual.

i.e.

```
class A
{       };

class B : virtual public A
{       };

class C : virtual public A
{       };

class D : public B, public C
{       };
```

as shown in above code, we should derive class B and class C from class A virtually. Such type of inheritance is called virtual inheritance.

```
class B : virtual public A      //Virtual inheritance


class C : virtual public A      //Virtual inheritance
```

If we declare base class [**A**]as virtual then members of A will be inherited into class B and C.if we derived class D from B and C then memebrs of A will not inherit from class B and C rather only non inherited members of B and C will be inherited into D. Then what about members of class A?

Members of class A will be Inherited into class D directcly.

Runtime Polymorphism

```
class Base
{
private:
    int num1;
    int num2;
public:
    Base( void ) : num1( 10 ), num2( 20 )
    {       }
    Base( int num1, int num2 ) : num1( num1 ), num2( num2 )
    {       }
    void showRecord( void )
    {
        cout<<"Num1 :        "<<this->num1<<endl;
        cout<<"Num2 :        "<<this->num2<<endl;
    }
    void printRecord( void )
    {
        this->showRecord();
    }
};
```

```
class Derived : public Base
{
private:
    int num3;
public:
    Derived( void ) : num3( 30 )
    {       }
    Derived( int num1, int num2, int num3 ) : Base( num1, num2 ), num3( num3 )
    {       }
    void printRecord( void )
    {
        Base::showRecord();
        cout<<"Num3 :       "<<this->num3<<endl;
    }
    void displayRecord( void )
    {
        this->printRecord();
    }
};
```

Lets revise all the concepts using above program.

```
1. Base* ptr = new Base();

ptr->showRecord();

ptr->printRecord();

ptr->Derived::printRecord();

ptr->displayRecord();

delete ptr.
```

```
3.Derived derived( 500,600,700);
 Base base;
 base = derived;
 base.printRecord();

  ptr->displayRecord();
  delete ptr


4.Base base( 500, 600 );
  Derived derived;
  derived = base;
  derived.printRecord();

5.Derived* ptrDerived = new Derived( );
 ptrDerived->printRecord();
 Base* ptrBase = ( Base* )ptrDerived;
 ptrBase->printRecord();


6.Base* ptrBase = new Derived( );
 ptrBase->printRecord();
```

```
7.Derived derivedInstance( 500,600, 700);

  Derived& derivedReference = derivedInstance;

  Base& baseReference = ( Base& )derivedReference;

  baseReference.printRecord();


8.Base* ptrBase = new Derived( );

  ptrBase->printRecord();

  Derived* ptrDerived = ( Derived* ) ptrBase;

  ptrDerived->printRecord();
```

Virtual function

If we want to write geeric program or maintainable program then we should use upcasting. In case of upcasting , function gets called depending on type of pointer.If we want to call function depending on type of object rather than type of pointer then we should declare function in base class as a virtual.

In case of upcasting, if we want to call function of derived class then we should declare function in base class as virtual.

What is virtual function?

A member function of a class, which gets called depending on type of object rather than type of pointer is called virtual function.

In other words, member function of derived class which is designed to call using reference or pointer of base class is called virtual function.

According to the definition of virtual function, it is designed to call using either pointer or reference of base class only. Virtual functions are not designed to call using object.

Virtual member functions are designed to call using base class pointer or reference only. Static member functions are designed to call using class name only. Since static member functions are not designed to call using pointer or reference, we can not declare sta member function as virtual.

If class contains at least one virtual function then such class is considered as polymorphic class. If signature of base class and derived class member function is same and if function in base class is virtual then derived class function is by default considered as virtual, hence it is optional to use virtual keyword with member function in derived class. In short, if base class is polymorphic then every derived class is considered as polymorphic.

```
class Product
{
private:
    string name;
    float price;
public:
    virtual void acceptRecord( void )    {       }
    virtual void printRecord( void )     {       }
};
class Book : public Product
{
private:
    int pageCount;
public:
    void acceptRecord( void )    {       }
    void printRecord( void )     {       }
};
class Tape : public Product
{
private:
    int playTime;
public:
    void acceptRecord( void )    {       }
    void printRecord( void )     {       }
};
```

Process of re-implementing virtual function of base class inside derived class with sam signature is called function overriding. For function overriding signature of base class and derived class function must be same and function in base class must be virtual.

Constructor and destructor do not inherit into the derived class, hence we can not override it. Since, static member function can not be declared as virtual, we can not override it in derived class.

If we give call to the virtual function using pointer or reference then it is considered as late binding.

If we give call to the non-virtual function using pointer or reference then it is considered as early binding.

If we give call to any virtual or non-virtual function using object then it is considered as early binding.

Consider the following program.

```
int menu_list( void )
{
    int choice;
    cout<<"0.Exit"<<endl;
    cout<<"1.Book"<<endl;
    cout<<"2.Tape"<<endl;
    cout<<"Enter choice      :        ";
    cin>>choice;
    return choice;
}
```

```
int main( void )
{
    int choice;
    while(( choice = menu_list( ) ) != 0 )
    {
        Product* ptrProduct = NULL;
        switch( choice )
        {
        case 1:
            ptrProduct = new Book();
            break;
        case 2:
            ptrProduct = new Tape();
            break;
        }
        if( ptrProduct != NULL )
        {
            ptrProduct->acceptRecord();
            ptrProduct->printRecord();
        }
    }       return 0;
}
```

Early binding and late binding:

Consider the following program and decide whether it is early binding or late binding.

```
class Base
{
public:
    virtual void F1( void ){    }
    virtual void F2( void ){    }
    virtual void F3( void ){    }
    void F4( void ){    }
    void F5( void ){    }
};
class Derived : public Base
{
public:
    virtual void F1( void ){    }
    void F2( void ){    }
    void F4( void ){    }
    virtual void F5( void ){    }
    virtual void F6( void ){    }
};
```

1. Basebase;

base.F1();

base.F2();

base.F3();

base.F4();

base.F5();

base.F6();

2.Base* ptrBase = **new**Base();

ptrBase->F1();

ptrBase->F2();

ptrBase->F3();

ptrBase->F4();

ptrBase->F5();

ptrBase->F6();

3.Base* ptrBase = **new**Derived();

ptrBase->F1();

ptrBase->F2();

ptrBase->F3();

ptrBase->F4();

ptrBase->F5();

ptrBase->F6();

4.Derived* ptrDerived = **new**Derived();

ptrDerived->F1();

ptrDerived->F2();

ptrDerived->F3();

ptrDerived->F4();

ptrDerived->F5();

ptrDerived->F6();

Algorithm to decide early binding and late binding

```
//Check Data Type of caller[ Base class or Derived class]
if( function is not exist in called Data type )
{
        Compiler error : Function is not a member of caller data type
}
else    //if Function is exist
{
        //Check Type of caller[ Object, poinyter or reference ]
        if( caller is object )
                Early binding : Caller type [ if not exist then inherited ] function will call.
        else    //If caller is pointer or reference
        {
                //Check type of function[ virtual or non virtual ]
                if( function is non virtual )
                        Early binding : Caller type [ if not exist then inherited ] function will call.
                else
                        Late binding : Object type [ if not exist then inherited ] function will call.
        }

}
```

Virtual function pointer and table

If class contains virtual function(s) then compiler implicitly creates a table which stores address of virtual function(s) declared inside the class. Such table is called virtual function table or vf-table or v-table.

In short, a table which stores address of virtual function declared inside class is called v-table.

To store address of virtual function table, compiler adds one hidden pointer as data member inside class. It is called virtual function pointer or vf-pointer or v-ptr.

In short, a pointer which stores address of v-table is called virtual function pointer.

If class contains virtual function then v-ptr gets added inside the class and hence size of the object gets increased.

V-table and v-ptr inherits into the derived class.In C++, if we create object of a class then constructor gets called. i.e constructor is not designed to call depending on type of pointer.

Virtual functions are designed to call using pointers. Since constructors are not desigend to call using pointer, we cannot declare constructor as virtual.

Initializing v-ptr, i.e storing address of v-table into v-ptr is a job of constructor. i.e virtual functions can execute their body after calling constructor. Hence we cannot declare constructor as a virtual.

We cannot declare constructor as virtual but we can declare destructor as a virtual.

In case of upcasting, to call derived class destructor first, it is necessary to declare destructor in base class as a virtual.

Consider the following code:

```
class Base
{
    int* ptr1;
public:
    Base( void ) : ptr1( new int[ 3 ] )
    {   }
    virtual ~Base( void )
    {
        delete this->ptr1;
    }
};
class Derived : public Base
{
    int* ptr2;
public:
    Derived( void ) : ptr2( new int[ 5 ] )
    {   }
    ~Derived( void )
    {
        delete this->ptr2;
    }
};
```

C++ - Hierarchy Notes

Only in case of inheritance, we need to declare function in base class as a virtual. In C++, every class is not designed for inheritance, hence function or destructor is not virtual by default.

Difference between function overloading and overriding.

Function Overloading	**Function Overriding**
Compile time polymorphism is achieved using function overloading.	Run time polymorphism is achieved using function overriding.
In this case function signature should not be same	In this case function signature must be same.
No keyword is required.	Function in base class must be virtual.
Return type is not considered.	Return type is considered
Implementation is based on mangled name.	Implementation is based on v-table.
It doesn't affect on size of object.	It affects on size of object.
Fuctions must be exist in same scope	Functions must be exist in base and derived class.

RTTI

RTTI and**advanced type casting** operators are features of advanced C++, but now it is considered as part of standard C++.

Header File Name : typeinfo

```
namespace std
{
      class type_info
      {
      protected:
          const char *__name;
      private:
              type_info(const type_info& other);    //Copy Constructor
              type_info& operator=(const type_info& other);
      public:
              const char* name() const; // Returns string
              bool operator==(const type_info& other) const;
              bool operator!=(const type_info& other) const;
              virtual ~type_info();
      };//End of type_info class

}//end of namespace
```

As shown in above code **type_info** is *class* which is declared in **std***namespace* and std namespace is declared in **typeinfo** *header file*.

If copy constructor or assignment operator function is private then we cannot create copy of the object inside non member function.

If we want to find out name of type of object at runtime, then we should use **typeid** operator.

typeid operator returns reference of type_info object which is constant object.

As shown in type_info class, copy constructor and assignment operator function of type_info class is private, hence we can not create copy of type_info class object in our program. To avoid copy operation we should use reference. Consider the following code.

```
#include<typeinfo>  //For type_info
#include<string>
using namespace std;

int main( void )
{
    int number = 10;

    const type_info& typeReference = typeid( number );

    string typeName = typeReference.name(); //int

    return 0;
}
```

As shown in above code, **name( )** is a non static method of type_info class, which returns name of type of object.

"This process of getting type of object object at runtime is called **RTTI**".

RTTI is not designed for simple types, It is designed for polymorphic type.

If we want to use name() method only once then we can use following way:

```
int number = 10;

cout<<"Type :   "<<typeid(number).name()<<endl;
```

RTTI is not designed to find out typename of simple objects, it is designed to work with polymorphic types.

In case of upcasting, if we want to find out information about true type of object then we should use RTTI.

Consider following code:

RTTI

```
class Base  //Non Polymorphic
{
public:
    void printRecord( void )
    {   }
};//End of Base class

class Derived : public Base
{
public:
    void printRecord( void )
    {   }
};//End of Derived class
```

Let us use RTTI with Base class pointer and object

```
Base* ptrBase = new Base( );

//On Linux
cout<< typeid( ptrBase ).name()<<endl;   //P4Base
//On Windows
cout<< typeid( ptrBase ).name()<<endl;   //class Base*

//On Linux
cout<< typeid( *ptrBase ).name()<<endl; //4Base
//On Windows
cout<< typeid( *ptrBase ).name()<<endl; //class Base
```

Let us use the RTTI with Derived class pointer and object.

```
Derived* ptrDerived = new Derived( );

//On Linux
cout<< typeid( ptrDerived ).name()<<endl;   //P7Derived
//On Windows
cout<< typeid( ptrDerived ).name()<<endl;   //class Derived*

//On Linux
cout<< typeid( *ptrDerived ).name()<<endl;  //7Derived
//On Windows
cout<< typeid( *ptrDerived ).name()<<endl;  //class Derived
```

Let us use the RTTI with upcasting

```
Base* ptrBase = new Derived( ); //Upcasting

//On Linux
cout<< typeid( ptrBase ).name()<<endl;  //P4Base
//On Windows
cout<< typeid( ptrBase ).name()<<endl;  //class Base*

//On Linux
cout<< typeid( *ptrBase ).name()<<endl; //4Base
//On Windows
cout<< typeid( *ptrBase ).name()<<endl; //class Base
```

If you observe above code carefully then you will realize that type of object is Derived but it is showing Base.

In case of upcasting if we want to find out true type of object then base class must be polymorphic i.e. we should declare at least one function as virtual or pure virtual.

Let's declare base class as a polymorphic.

```
class Base  //Polymorphic
{
public:
    virtual void printRecord( void )
    {   }
    virtual ~Base( void )
    {   }
};//End of Base class

class Derived : public Base
{
public:
    void printRecord( void )
    {   }
};//End of Derived class
```

If base class is polymorphic then all its derived classes are considered as polymorphic. Now let's use RTTI with upcasting.

```
Base* ptrBase = new Derived( ); //Upcasting

//On Linux
cout<< typeid( ptrBase ).name()<<endl;   //P4Base
//On Windows
cout<< typeid( ptrBase ).name()<<endl;   //class Base*

//On Linux
cout<< typeid( *ptrBase ).name()<<endl; //7Derived
//On Windows
cout<< typeid( *ptrBase ).name()<<endl; //class Derived
```

RTTI

Using NULL pointer, If we try to find out true type of object then C++ runtime throws **bad_typeid** exception. Let us see following code:

```
try
{
    Base* ptrBase = NULL;

    cout<< typeid( ptrBase ).name()<<endl;   //P4Base

    cout<< typeid( *ptrBase ).name()<<endl; //throws exception
}
catch (bad_typeid& e)
{
    cout<<e.what()<<endl;    //std::bad_typeid
}
```

Casting Operator

1. **reinterpret_cast** :
   It is used to convert pointer of any type into pointer of any other type. In other words, if you want to do type conversion between incompatible types then we should use reinterpret_cast operator.

```
Syntax -- reinterpret_cast< type-id >( expression )
```

In C++, if we want to access state of private datamembers outside the class then we can use any one of the following feature:

i. Member function
ii. Friend function
iii. Pointer

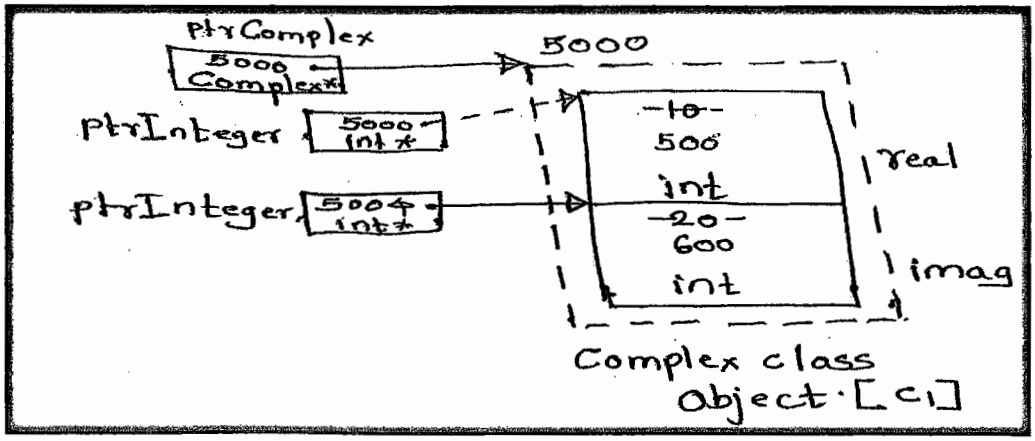
Let's see how to use pointer to access state of private members outside the class.

```
Complex c1;
cout<<c1;    //10, 20

Complex* ptrComplex = &c1;
//int* ptrInteger = (int*)ptrComplex;   //Old C Style
int* ptrInteger = reinterpret_cast<int*>( ptrComplex );

*ptrInteger = 500;
ptrInteger = ptrInteger + 1;
*ptrInteger = 600;

cout<<c1;    //500, 600
```



Casting Operator

Consider another example:

```
ofstream out;
out.open("Employee.data", ios::out | ios::binary );
if( out.is_open())
{
    Employee emp("Sandeep Kulange",33,25000.50f);
    out.write(reinterpret_cast<const char*>( &emp ) , sizeof( Employee ) );
}
else
    cerr<<"File io error"<<endl;
```

In above code we are trying to convert Employee pointer into character pointer(string).

2. **const_cast** :
   If we want to convert pointer to constant object into pointer to non-constant object or reference to constant object into reference to non-constant object then we should use const_cast operator.

```
Syntax -- const_cast< type-id >( expression )
```

Lets revise some points

- Using non constant, object we can access constant as well as non constant memebr function of the class.
- Using constant object, we can access only constant member functions of the class.
- If type of this pointer is (**ClassName* const this**) then inside member function we can modify state of the current object.
- If type of this pointer is (**const ClassName* const this**) then inside member function we can not modify state of the current object.
- If we declare member function as const then it gets this pointer (**const ClassName* const this**) like this.
- Inside non constant member function we can access constant as well as non constant member functions of the class
- Inside constant member function we can access only constant member functions of the class.

Casting Operator

If we want to access non constant member function inside constant member function then we should use following technique:

```
class Test
{
public:
        void showRecord( /* Test* const this */ )
        {
                cout<<"Inside showRecord"<<endl;
        }
        void printRecord( /* const Test* const this */ )const
        {
                //this->showRecord();   //Not Allowed
                //Test* const ptr = (Test* const)this;  //Old C Style
                Test* const ptr = const_cast<Test* const>(this);        //New Way
                ptr->showRecord();
        }
};
int main()
{
        const Test t;
        t.printRecord();
        return 0;
}
```

static_cast :

If we want to do type conversion between compatible types then we should use static_cast operator.

Consider the following code:

```
double num1 = 10.5;

//int num2 = ( int )num1;    //Old C Style

int num2 = static_cast<int>( num1 );     //New Way

cout<<"Num2 :    "<<num2<<endl;   //10
```

Casting Operator

```
class Base
{
private:
    int num1;
    int num2;
public:
    void setNum1(int num1)
    {
        this->num1 = num1;
    }
    void setNum2(int num2)
    {
        this->num2 = num2;
    }
    void printRecord( void )
    {
        cout<<"Num1 :   "<<this->num1<<endl;
        cout<<"Num2 :   "<<this->num2<<endl;
    }
};
```

It is mainly designed to do downcasting.

"In case of non polymorphic type, if we want to do downcasting then we should use static_cast operator."

Lets consider the following code to understand static_cast operator.

In case of Inheritance, members of base class inherits into the derived class hence every derived class object can be considered as base class object.

Since every derived class object can be considered as base class object, it is possible to store address of derived class object into base class pointer.

```
class  Derived : public Base
{
private:
    int num3;
public:
    void setNum3( int num3 )
    {
        this->num3 = num3;
    }
    void printRecord( void )
    {
        Base::printRecord();
        cout<<"Num3 :   "<<this->num3<<endl;
    }
};
```

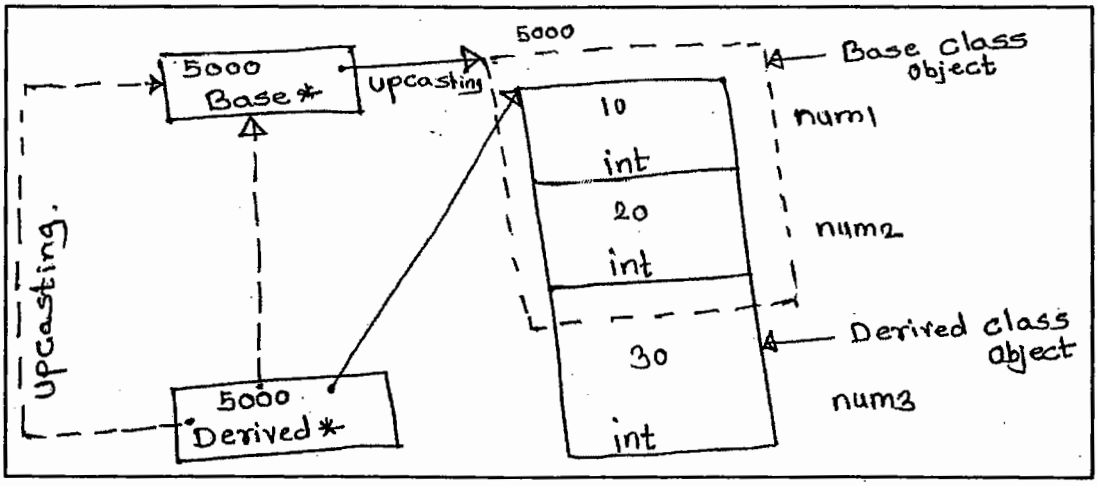
Casting Operator

```
Derived* ptrDerived = new Derived();
ptrDerived->setNum3(30);

//Base* ptrBase = ( Base* ) ptrDerived; //Upcasting
Base* ptrBase = ptrDerived; //Explicit typecasting is optional

ptrBase->setNum1(10);
ptrBase->setNum2(20);

ptrDerived->printRecord();
```



As shown in above diagram, process of converting pointer of derived class into pointer of base class is called as **upcasting**.

If we convert reference of derived class object into reference of base class then also it is called as **upcasting**.

```
Derived derivedInstance;

Derived& derivedReference = derivedInstance;

Base& baseReference = derivedReference; //Upcasting using reference
```

In case upcasting, using base class pointer or reference we can access only members of base class. In this case if we want to access data members or member function of derived class which is not exist in base class then we should do downcasting.

Casting Operator

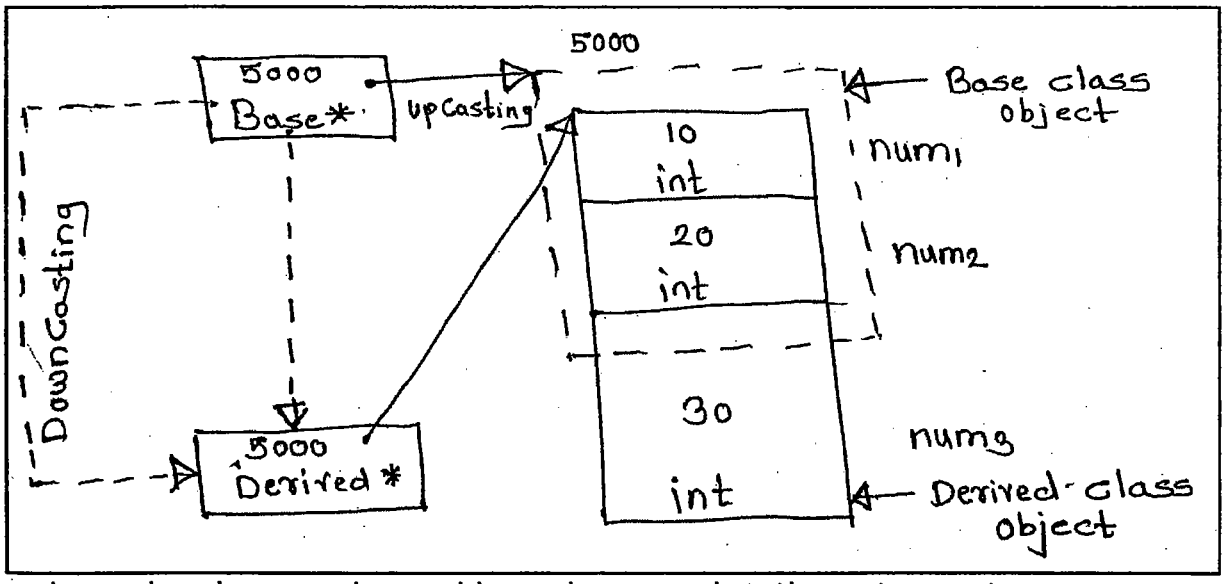
```
Base* ptrBase = new Derived();

ptrBase->setNum1(10);
ptrBase->setNum2(20);

Derived* ptrDerived = static_cast<Derived*>( ptrBase ); //Downcasting

ptrDerived->setNum3(30);

ptrDerived->printRecord();
```



Since base class is non polymorphic we have used static_cast operator.

Since members of derived class do not inherit into the base class, every base class object can not be considered as derived class object.

e.g. Every rectangle is a shape but every shape is not a rectangle.

If base class object can not be considered as derived class object then it is not possible to store address of base class object into derived class pointer.

Casting Operator

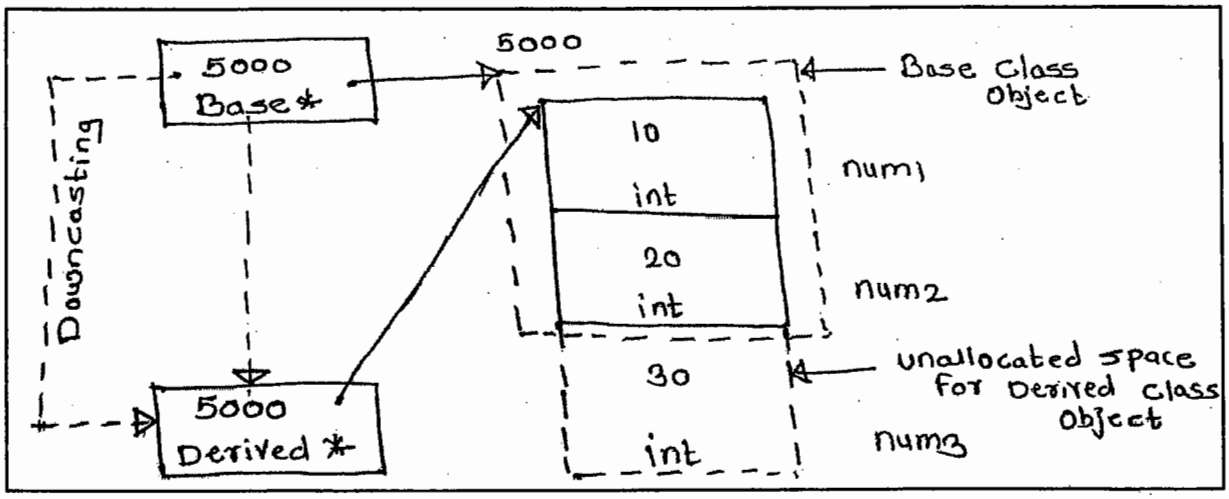
```
Base* ptrBase = new Base();

ptrBase->setNum1(10);
ptrBase->setNum2(20);

Derived* ptrDerived = static_cast<Derived*>( ptrBase ); //Downcasting

ptrDerived->setNum3(30);

ptrDerived->printRecord();
```



According to the concept, above conversion is invalid but *"static_cast operator do not check wheather type conversion is valid or not. It only checks inheritance relationship between source and destination type"*.

Points to remember:

- To do type conversion between numeric types we should use static_cast operator.
- In case of non polymorphic type, to do downcasting, we should use static_cast operator.
- static_cast operator checks inheritance relationship at compile time.
- It do not check wheather type conversion is valid or not.

3. **dynamic_cast** :

Casting Operator

During type conversion if we want to check both i.e. inheritance relationship and type

```
class Base  //Now Base class is polymorphic class
{
private:
    int num1;
    int num2;
public:
    void setNum1(int num1)
    {
        this->num1 = num1;
    }
    void setNum2(int num2)
    {
        this->num2 = num2;
    }
    virtual void printRecord( void )
    {
        cout<<"Num1 :   "<<this->num1<<endl;
        cout<<"Num2 :   "<<this->num2<<endl;
    }
};
```

conversion is valid or not then we should use dynamic_cast operator.

To use dynamic_cast operator base class must be polymorphic i.e. base must contain at least on virtual function.In other words, *"in case of polymorphic type, if we want to*

```
class  Derived : public Base
{
private:
    int num3;
public:
    void setNum3( int num3 )
    {
        this->num3 = num3;
    }
    void printRecord( void )
    {
        Base::printRecord();
        cout<<"Num3 :   "<<this->num3<<endl;
    }
};
```

do downcasting then we should use dynamic_cast operator".

```
Base* ptrBase = new Base();

ptrBase->setNum1(10);
ptrBase->setNum2(20);

Derived* ptrDerived = dynamic_cast<Derived*>( ptrBase ); //Downcasting
if( ptrDerived != NULL )
{
    ptrDerived->setNum3(30);
    ptrDerived->printRecord();
}
else
    cout<<"NULL Pointer"<<endl;
//Output : NULL Pointer
```

```
try
{
    Base baseInstance;
    Base& baseReference = baseInstance;
    Derived& derivedReference = dynamic_cast<Derived&>(baseReference);//throws bad_allo
    derivedReference.printRecord();
}
catch (exception& ex)
{
    cout<<ex.what()<<endl;
}
//Output : std::bad_cast
```

In case of pointer if dynamic_cast operator fail to do type conversion then it returns **NULL**.

Every base class object cannot be considered as derived class object therefore cannot store address of base class object into derived class pointer. Hence in above code dynamic_cast operator returns NULL.

As shown in above code, in case of reference, If dynamic_cast operator fail to do type conversion then it throws **bad_cast** exception.

We should always remember that, in case of only upcasting there might be chances to do downcasting. In rest of the condition downcasting may fail.

Casting Operator

dynamic_cast operator implicitlu use RTTI(typeid) to verify type conversion.

In following code, now there is no harm. After downcasting derived class pointer can store address of derived class object only.

```
Base* ptrBase = new Derived();  //Upcasting.

ptrBase->setNum1(10);
ptrBase->setNum2(20);

Derived* ptrDerived = dynamic_cast<Derived*>( ptrBase ); //Downcasting
if( ptrDerived != NULL )
{
    ptrDerived->setNum3(30);
    ptrBase->printRecord(); //late binding
}

else
    cout<<"NULL Pointer"<<endl;
//Output : 10, 20, 30
```

Points to remember:

- In case of polymorphic type, to do downcasting, we should use dynamic_cast operator.
- dynamic_cast operator checks inheritance relationship at run time.
- During type conversion, it checks wheather conversion is valid or not.
- In case of pointer if dynamic_cast operator fail to do type conversion then it returns NULL.
- In case of reference if dynamic_cast operator fail to do type conversion then it throws bad_cast exception.

Abstract Class

If implementation of base class is partially complete and without changing its implementation if we want to add new functionality in it then we should create its derived class.

If implementation of base class member function is partially complete then we should declare function in base class as virtual and to provide complete implementation we should override it in derived class. Consider the following example.

```
class Object
{
public:
    virtual string toString()
    {
        string typeName = typeid( *this ).name();
        return typeName;
    }
    virtual ~Object( void ){    }
};
class Complex : public Object
{
private:
    int real, imag;
public:
    Complex( void ) : real( 10 ), imag ( 20 )
    {    }
    virtual string toString()   //Overriding method here
    {
        char text[ 50 ];
        sprintf(text,"[Real = %d , Imag = %d]",this->real,this->imag);
        return string(text);
    }
};
```

As shown in above code, if we do not override toString member function in derived class then it will return fully qualified name of the class. We want to return state of Complex class object in string format hence it is necessary to override toString in Complex class.

If implementation of base class member function is logically totally unknown/incomplete in base class then we can declare such member function as pure virtual and to provide complete implementation we should override it in derived class.

Abstract Class

```
class Shape
{
protected:
    float area;
public:
    Shape( void ) : area( 0 )
    {   }

    virtual void calculateArea( void ) = 0; //Pure virtual function

    float getArea( void )
    {
        return this->area;
    }
    virtual ~Shape( void ){ }
};
```

"If we equate any virtual function to zero then it is called as pure virtual function".

We cannot equate any function to zero. To equate function to zero, it must be virtual.

We cannot provide body for the pure virtual function.

According to OOPs concept a member function of class which is having body is called *concrete method* and a function which is having body is called *abstract method.*

In C++ abstract keyword is not given. Pure virtual function can be called as abstract method.

"*If class contains at least one pure virtual function then such class is called as abstract class*".

Since abstract class contains incomplete implementation, *we can not create object of abstarct class.*

We cannot instantiate abstract class but we can create either pointer or reference of abstract class.

If we declare function in base class as virtual then overriding it in derived class is optional but if we declare function in base class as pure virtual then it is mandatory to override it in derived class.

Abstract Class

If we do not override pure virtual function in derived class then derived class will be considered as abstract class.If we want to instantiate derived class then it is compulsory to override pure virtual function in derived class. Lets Consider the two derived classes Shape.

```
class Rectangle : public Shape
{
private:
    float length;
    float breadth;
public:
    Rectangle( void ) : length( 0 ), breadth( 0 )
    {   }

    void setBreadth(float breadth)
    {
        this->breadth = breadth;
    }

    void setLength(float length)
    {
        this->length = length;
    }

    void calculateArea( void )  //Overriding pure virtual function of Base class
    {
        this->area = this->length * this->breadth;
    }
};
```

In accept_record function we will create object of Rectangle and then we will set value for length and breadth. To calculate the area, we can call calculateArea() function in main or at the end of accept_record function.

We have declared getArea() function in base class, it will inherited in derived class and can be used to get area from rectangle object.

Abstract Class

```
class Math  //utility class
{
public:
    static const float PI;
public:
    static float power( float base, int index )
    {
        float result = 1;
        for( int count = 1; count <= index;  ++ count  )
            result = result * base;
        return result;
    }
};
```

Math is utility class written to calculate area of circle.
Now lets implement class Circle to calculate its area:

```
class Circle : public Shape
{
private:
    float radius;
public:
    Circle( void ) : radius( 0 )
    {  }

    void setRadius(float radius)
    {
        this->radius = radius;
    }

    void calculateArea( void )  //Overriding pure virtual function of Base class
    {
        this->area = Math::PI * Math::power(this->radius, 2 );
    }
};
```

In accept_record function we will create object of Circle and then we will set value for radius. To calculate the area, we can call calculateArea() function in main or at the end of accept_record function.

Inherited function getArea() will be used to get area from the circle object.

Abstract Class

By defining pure virtual function in base class we can maintain same method design signature in all the derived classes which will be helpful to maintainable code.

Let's complete the above program and discuss how to achieve runtime polymorphism in C++

Polymorphism is an ability of any object to use same interface to perform different operation or behavior. We should use polymorphism to write maintainable code.

To write maintainable code we should use upcasting. In case of upcasting we can call only those function of derived class which are overrided from base class. If we want force all the derived classes to keep same function signature then we should declare all the functions in base class as pure virtual[abstract].

To minimize the complexity we will take help of factory class/ method. A class which hides object creation process from end user is called factory class. It is a creational design pattern.

```
enum ShapeType
{
    EXIT, RECTANGLE,CIRCLE
};

class ShapeFactory
{
public:
    static Shape* getInstance( ShapeType shapeType )
    {
        switch( shapeType )
        {
        case RECTANGLE:
            return new Rectangle();
            break;
        case CIRCLE:
            return new Circle();
            break;
        }
        return NULL;
    }
};
```

Let's implement main function so that it will make picture more clear.

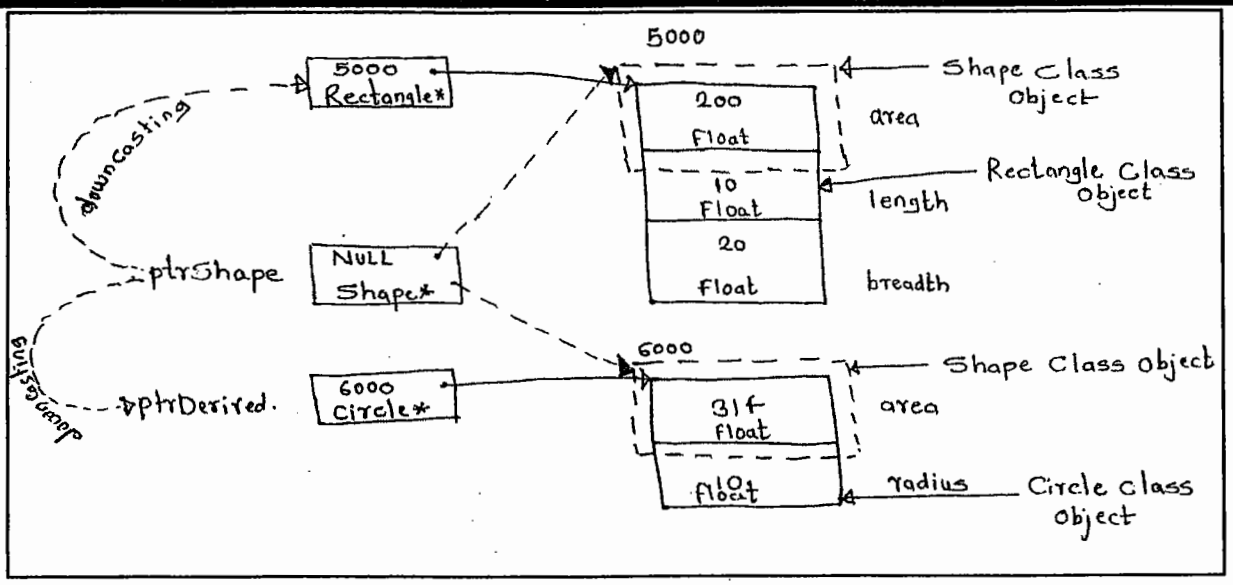
Abstract Class

```
void accept_record( Shape* ptrShape )
{
        //TODO
}
void print_record( Shape* ptrShape )
{
        //TODO
}
ShapeType menu_list( void )
{
        int choice;
        cout<<"0.Exit"<<endl;
        cout<<"1.Rectangle"<<endl;
        cout<<"2.Circle"<<endl;
        cout<<"Enter choice     :       ";
        cin>>choice;
        return ShapeType( choice );
}
int main()
{
        ShapeType choice;
        while( ( choice = ::menu_list()) != EXIT)
        {
                Shape* ptrShape = ShapeFactory::getInstance(choice);
                if( ptrShape != NULL )
                {
                        ::accept_record(ptrShape);
                        ::print_record(ptrShape);
                        delete ptrShape;
                }
        }
        return 0;
}
```

According to the code written in main function, ptrShape (Base class pointer) can store address of any one of the derived class object(Rectangle/Circle) one at a time.

Code written in main function will work for any object which is derived from Shape class. Before discussing implementation of accept_record and print_record function let's see pictorial view.

Abstract Class



Lets Consider Rectangle object. Area is depends on value of length and breadth. To set length and breadth we should use setLength and setBreadth function.

Using base class pointer we can not access undeclared member function of base class which is a member function of derived class.Hence to access setter functions of Rectangle class we need to do downcasting. For the Circle object also we should do same thing.

```
void accept_record( Shape* ptrShape )
{
    if( dynamic_cast<Rectangle*>(ptrShape) != NULL )
    {
        Rectangle* ptrRectangle = dynamic_cast<Rectangle*>(ptrShape);
        ptrRectangle->setLength(10);
        ptrRectangle->setBreadth(20);
    }
    else
    {
        Circle* ptrCircle = dynamic_cast<Circle*>(ptrShape );
        ptrCircle->setRadius(10);
    }
    ptrShape->calculateArea();  //Runtime polymorphism
}

void print_record( Shape* ptrShape )
{
    string typeName = typeid(*ptrShape).name();
    cout<<"Area of instance of "<<typeName<<" is "<<ptrShape->getArea()<<endl;
}
```

Abstract Class

In accept_record function, first dynamic_cast operator is used to check, is it storing address of Rectangle or Circle class object and then it is used to do downcasting.

Using dynamic_cast operator, apart from type conversion, we can find out at runtime whether base class pointer is storing address of which derived class object.

ptrShape->calculateArea(); Function call will be resolve at runtime. If ptrShape stores address of Rectangle object then above function call will calculate area of rectangle and if it stores address of Circle class object then it will calculate area of circle. Depending on type of object behaviour of calculateArea() will be decided at runtime.

"At runtime, a*n ability of objects of different types to use same interface to perform different operation is called runtime polymorphism*".

If class contains all pure virtual function then such class is called is called as pure abstract class. Pure abstract class is also called as interface.

```
class LinkedList
{
public:
    virtual void addFirst( int data ) = 0;
    virtual void addLast( int data ) = 0;
    virtual void removeFirst( void ) = 0;
    virtual void removeLast( void ) = 0;
    virtual void clear( void ) = 0;
};
```

In above code, LinkedList is considered as pure abstract class or interface.

Interface inheritance:

At the time of inheritance, if base type and derived type is interface then it is called as interface inheritance. Consider the following code.

Abstract Class

```
class A
{
public:
    virtual void f1( void ) = 0;

    virtual void f2( void ) = 0;
};
class B : public A  //Interface Inheritance
{
public:
    virtual void f3( void ) = 0;
};
```

In above code both A and B are interfaces.

Interface implementation inheritance:

At the time of inheritance, if base type is interface and derived type is class then it is called as interface implementation inheritance. Consider the following code.

```
class A
{
public:
    virtual void f1( void ) = 0;

    virtual void f2( void ) = 0;
};
class B : public A  //Interface implementation Inheritance
{
public:
    virtual void f1( void )
    {
        //TODO
    }

    virtual void f2( void )
    {
        //TODO
    }
};
```

Implementation inheritance:

At the time of inheritance, if base and derived types are classes then such type of inheritance is called implementation inheritance. Consider the following code.

Abstract Class

```
class A
{
public:
    void f1( void )
    {   }
};
class B : public A  //Implementation Inheritance
{
public:
    void f2( void )
    {   }
};
```

If we declare virtual function inside the class then compiler create v-table for the class and to store its address compiler adds v-ptr inside the class.

Initializing v-ptr, i.e. storing address of v-table into v-ptr is a job of constructor.

If class contains pure virtual function then also v-table gets generated for the class. Hence to initialize v-ptr, abstract class needs constructor. As per application requirement, we can write

```
class Node
{
protected:
    Node* next;
public:
    Node( void ) : next( NULL )
    {   }
    friend class LinkedList;
    virtual ~Node( void ){  }
};
```

constructor inside the class or we can rely on default constructor.

"At least to initialize v-ptr, abstract class needs constructor."

Let us see, how to create heterogeneous linked list using inheritance.

```
class IntegerNode : public Node
{
private:
    int data;
public:
    IntegerNode( int data = 0 ) : data( data )
    {   }
    friend class LinkedList;
};
class StringNode : public Node
{
private:
    string data;
public:
    StringNode( string data = "" ) : data( data )
    {   }
    friend class LinkedList;
};
```

Abstract Class

```
class LinkedList
{
private:
    Node* head;
    Node* tail;
public:
    LinkedList( void ) : head( NULL ),tail(NULL)
    {   }
```

```
void addLast( int data )
{
    Node* newnode = new IntegerNode(data);
    if( head == NULL )
        head = newnode;
    else
        tail->next = newnode;
    tail = newnode;
}
void addLast(string data )
{
    Node* newnode = new StringNode(data);
    if( head == NULL )
        head = newnode;
    else
        tail->next = newnode;
    tail = newnode;
}
```

```
void printList( void )
{
    if( head != NULL )
    {
        Node* trav = head;
        while( trav != NULL )
        {
            if( dynamic_cast<IntegerNode*>(trav) != NULL )
            {
                IntegerNode* node =  dynamic_cast<IntegerNode*>(trav);
                cout<<node->data<<" ";
            }
            else
            {
                StringNode* node =  dynamic_cast<StringNode*>(trav);
                cout<<node->data<<" ";
            }
            trav = trav->next;
        }
        cout<<endl;
    }
}
~LinkedList( void ) {   /*TODO*/   }
```

Data Structure:

- Structure of Data
  - The organization of data in memory.
  - Operations performed on that data.
- Types of data structures:
  - Linear - Array, Stack, Queue, Linked List.
  - Non-Linear - Tree, Graph
  - Associative:- data is stored in key value pair e.g. Hash Table
- Usually data structures are represented as "Abstract Data Types".
- Example Data Structures:
  - Array, Stack, Queue, Linked List, Tree, Graph, Hash Table, etc.

Algorithm:

- An algorithm is a **finite set of instructions**, if followed, accomplishes a particular task.
- Algorithms are written to solve problems. One problem may have multiple solutions, so we need to decide which algorithm is best amongst them to solve a particular problem.
- There are two main measures to decide the **efficiency** of an algorithm: **time** and **space.**
- **Analysis or Performance of an algorithm** refers to the task of determining how much **computing time** (time) and **storage** (space) requires running to completion.

Space Complexity

- Space Complexity of an algorithm is **the amount of memory it needs to run to completion.**
- Space required by a program = **code space** (instructions resides in memory) + **data space** (space taken by variables and constants)
- The space required has two components:
  - **Fixed component:** it is independent of instance characteristics. It includes **instruction space, space required by simple variables, aggregate variables** and **constants.**
  - **Variable Component:** is depends on **instance characteristics** such as size of the input . It includes the space needed by **reference variables** and **recursion stack.**
- The space required **S(P)** of any algorithm/program P may therefore be written as: **S(P) = C + S(instance characteristics) = C + S_P(n);** where C is a constant and n is the input size.

```
int sum(int arr[],int n) //arr is an array of size n
{
      int s = 0;
      for(int i = 0; i < n; i++)
            a = s + arr[i];
      return s;
}
```

- **Space required = Code space + Space required by variables**

- $S_P(n)$ = Space required by variables
- $S_P(n)$ = n (for arr) + 1 (for n) + 1 (for s) + 1 (for i) + 2 for(0 and 1)
- $S_P(n)$ = n + 5
- $S_P(n)$ >= n

- **Space Complexity for Recursive Algorithms:**

```
int Rsum(int arr[], int n)//recursive: where arr is an array of
size n
{
      if(n <= 0)
            return 0;
      else
            return Rsum(arr,n-1) + arr[n-1];
}
```

- **Space required = code space + space required by variables + stack space.**
  - Recursion stack space includes space for **formal parameters, local variables** and **return address** = 1 (for n) + 1 (for pointer to A) + 1 (for return address)
  - The depth of recursion is = n+1
  - **S_P>= 3(n+1)**
- **Recursive algorithms have very high space complexity.**

Time Complexity

- Time Complexity of an algorithm is **the amount of computer time it needs to run to completion.**
- It is the sum of **compile time** (fixed components) + **run or execution time**(depends on instance characteristics).
- The execution time of a statement depends on **type of operations involved, values, the type of machine** and **the type of environment in which the statement is executed.**
- **Best Case Analysis:** If algorithm takes **minimum time** to solve the problem for given set of input, it is called as best case time complexity.
- **Worst Case Analysis:** If algorithm takes **maximum time** to solve the problem for a given set of input is is called as worst case time complexity, it is the special case of interest for the mathematics.
- **Average Case Analysis:** Input sequence which is neither best nor worst is called as average case. Average case analysis represents general behavior of an algorithm.
- In most of the cases average and worst cases has same magnitude.

- **Asymptotic Notations** are the mathematical tool to find time or space complexity of an algorithm without implementing it in programming language.
  - This measure is independent of machine specific constants.
  - Asymptotic notation is a way of describing major component of the cost of entire algorithm.
  - Primitive operation is the major component of the cost.

- Asymptotic notations is a way of describing major component of the cost of entire algorithm

- **Assumptions while doing complexity analysis:**
  - Actual cost of operation is not considered.
  - Abstract cost c is ignored: **$O(c.n^2)$ reduces to $O(n^2)$**
  - Only leading term of polynomial is considered: **$O(n^3 + n)$ reduces to $O(n^3)$**
  - Drop multiplicative or divisive constant if any: **$O(2n^2)$ and $O(n^2/2)$ both reduces to $O(n^2)$**
  - Various notations like **Big Oh (O), Big Omega (Ω)andBig Theta (Θ)** are used to describe the asymptotic running time of an algorithm.
  - Above notations are useful in algorithm analysis to find **best, worst** and **averagecomplexity** of algorithms in terms of **primitive operations.**

- **Big Oh notation (O):** It is used to define an **asymptotic upper bound for algorithm,** it means running time of algorithm cannot be more than its asymptotic upper bound for any random sequence of data.
  - **Definition:** A function **f(n)** is a member of **O(g(n))** or we write **f(n) = O(g(n))** or say **f(n) is of O(g(n))**, if there exists positive constants c and n_0 such that, **$0 <= f(n) <= c.g(n)$ for all $n >= n_0$.**
  - **$O(g(n)) = \{ f(n)$: there exists a positive constant c and n_0 such that $0 <= f(n) <= c.g(n)$ for all $n >= n_0 \}$.**
  - **O -notation** is used to describe an upper bound for the **worst case running time** of an algorithm.

- **Big Omega notation(Ω):** It is used to define an **asymptotic lower bound.**
  - **$\Omega(g(n)) = \{ f(n)$: there exists positive constants c and n_0, such that $0 <= c.g(n) <= f(n)$ for all $n >= n_0 \}$.**
  - **Ω-notation** is used to describe a lower bound for the **best case running time** of an algorithm.
  - **Insertion Sort:** is **$O(n^2)$** and is **$\Omega(n)$.**

- **Big Theta notation (Θ):** There is separate notation for **asymptotically tight bounds;Θ-notation** is used to define an **asymptotic tight bound.**
  - We say g(n) is a tight bound for f(n), if there exists positive constants c1,c2 and n_0 such that **$0 <= c1.g(n) <= f(n) <= c2.g(n)$** for all $n >= n_0$. It is denoted as **$f(n) = \Theta(g(n))$.**

- **Definition:** A function **f(n)** is a member of **Θ(g(n))** or we write **f(n) = Θ(g(n))** or we say **f(n) is of Θ(g(n))** if there exists a positive constants c and n_0 such that, **0 <= c1.g(n) <= f(n) <= c2.g(n)** for all n >= n_0.
- **Θ(g(n)) = { f(n): there exists positive constants c1,c2 and n_0 such that 0 <= c1.g(n) <= f(n) <= c2.g(n), for all n >= n_0}.**
- **f(n) is equal to g(n) within a constant factor, the function f(n) must be non-negative.f(n) = Θ(g(n)).**
- The function g is both, an upper as well as lower bound for f.
- Describes the growth of the function more accurately than **O** and **Ω.**
- The running time of an algorithm is **Θ(g(n)) iff (if and onlyif)**Its worst case running time is **O(g(n))** and Its best case running time is **Ω(g(n)).**

Analysis of Iterative Programs

- **O(1)**
  - time complexity of a function(or set of elements) is considered as O(1), if it doesn't contain a loop, recursion and call to any other non-constant time function.
  - **//set of non-recursive and non-loop statements**
  - e.g. **swap()** function has **O(1)** time complexity
  - A loop or recursion that runs a constant number of times is also considered as **O(1)**, e.g. the following loop is: O(1)

```
//here c is constant
for( int i = 1; i <= c ; i++ ){
      //some O(1) expressions
}
```

- **O(n)**
  - time complexity of a loop is considered as O(n) if the loop variable is **incremented/decremented** by a constant amount. For example following function has O(n) time complexity.

```
//here c is any positive integer constant
for( int i = 0 ; i <= n ; i += c ){
       //some O(1) expressions
}
OR
for(int i = n ; i > 0 ; i  -=  c ){
     //some O(1) expressions
}
```

- **O(n^c):**

- time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have **O(n^2)** time complexity.

```
for(int i = 0 ; i <= n ; i += c) //-> loop runs n times
{
     for(int j = 0 ; j <= n ; j += c ){ // -> loop runs n times
          //some O(1) expressions
      }
}
for(int i = n ; i > 0 ; i -= c ) // -> loop runs n times
{
     for(int j = i+1 ; j <= n ; j += c ){// -> loop runs n times
       //some O(1) expressions
      }
}
```

- for example **selection sort** and **insertion sort** have **O(n^2)** time complexity.

- **O(log n):**
  - time complexity of a loop is considered as **O(log n)** if the loop variables **divided/multiplied** by a constant amount.

```
for(int i = 0 ; i <= n ; i *= c ){
  //some O(1) expressions
}
for(int i = n ; i > 0 ; i /= c ){
  //some O(1) expressions
}
```

  - e.g. **binary search** has **O(log n)** time complexity.

- **O(log(log n)):**
  - time complexity of a loop is considered as **O(log(log n))** if the loop variables is **reduced/increased exponentially** by a constant amount.

```
//here c is a constant greater than 1
for(int i = 2 ; i <= n ; i = pow(i,c) ) {
    //some O(1) expressions
}
//here fun is sqrt or cuberoot or any other constant
root
for(int i = n ; i > 0 ; i = fun(i) ){
    //some O(1) expressions
}
```

- **How to combine time complexities of consecutive loops?**

  When there are consecutive loops, we calculate time complexity as sum of the time complexities of indivisual loops.

```
for(int i = 1 ; i <= m ; i += c ){
    //some O(1) expressions
}
for(int i = 1 ; i <= n ; i += c ){
    //some O(1) expressions
}
```

Time complexity of above code is **O(m) + O(n)** which is **O(m+n)**

If **m == n**, the time complexity becomes **O(2n)** which is **O(n)**.

SEARCHING TECHNIQUES:

1. Linear Search:

- It is also known as **sequential search**.
- Suppose **A** is an array of size **n**, we want to search an element from this array. Let us call this element as **key**.
- Linear Searching is very simple way of searching. Key element is compared with one by one all index location of A.
- Algorithm halts in two cases: key element is found or entire array is scanned.

```
int LinearSearch(int A[],int key)
{
   //A is an array of size n, and key is the element to be
search
   for( int index = 0 ; index < n ; index++ )
   {
        if A[index] == key
        then
             return index;
   }
   return -1;
}
```

- **Complexity Analysis:**

Best Case:

- Algorithm needs minimum number of comparisons if key element is found on very first location.
- In best case, size of input array does not matter. If key element is on the first position it takes only one comparison irrespective of array length. Hence the running time of linear search in best case is, **T(n) = 1.**

Worst Case:

- Algorithm performs maximum number of comparisons if key element is on the last location or it is not present at all. Entire array needs to be scanned.
- No. of comparisons linearly grow with the size of input. Hence the running time of linear search in worst case is, **T(n) = O(n).**

Average Case:

- Average case occurs when element is neither on first location nor on last. Key element may be near to beginning or may be near to end, or it may be somewhere near to middle. So on an average, algorithm does (n/2) comparisons.
- **T(n) = O(n/2) => O(n).**

2. Linear Search Recursive:

```
int RecLinearSearch(int A[], int index, int key)
{
   // A is the array of size n, key is the ele to be search
```

```
    if(   index == n )
          retrun -1;
    if( A[index] == key )
          return index;
    else
          returnRecLinearSearch(A,index-1,key);
}
```

3. Binary Search:

- Binary search is **efficient** than linear search.
- For binary search array must be sorted, which is not required in linear search.
- Binary search reduces search space by half in every iteration, whereas in linear search, search space is reduced by one in each iteration.
- If there are **n** elements in an array, binary search and linear search has to search among (n/2) and (n-1) elements respectively in second iteration.
- In third iteration, binary search has to scan only (n/4) elements, whereas linear search has to scan (n-2) elements. This shows that binary search would hit the bottom very quickly.

```
int BinarySearch(int A[],int left,int right,int key)
{
   while( left <= right )
   {
         mid = (left + right)/2;
         if( A[mid] == key )
               return mid;
         if( key < A[mid] )
               right = mid-1;
         else
               left = mid+1;
   }
   return -1;
}
```

- **Complexity Analysis:**

Best Case:

- On very first attempt, a key element is compared with middle element of an array. If key element is on middle position of array, algorithm does only one comparison, irrespective of size of array.
- Hence best case running time of algorithm would be, **T(n) = O(1).**

Worst Case:

- Search space is reduced by half in every iteration, so maximum $\log_2 n$ divisions are possible. If element is at leaf of the tree or it's not present at all, then algorithm does $\log_2 n$ comparisons, which are maximum. No. of comparisons grow in logarithmic order of input size, Hence, worst case running time of algorithm would be, $T(n) = O(\log_2 n)$

Average Case:

- Average case for binary search occurs when key element is neither on middle nor at the leaf level of the tree. On an average, it does half of the $\log_2 n$ comparisons, which will turn out as, **$T(n) = O(\log_2 n)$**

Recurrence Equation for Binary Search and Running time of the recurrence is $O(\log_2 n)$:
In each iteration, binary search does one comparison and creates the new problem of size n/2.
So recurrence equation of binary search is given as,
$T(n) = T(n/2) + 1, n > 1$
$T(n) = 1$, $n = 1$ i.e. only one comparison is needed when there is only one element in a array, that's the trivial case.

This is the boundary condition for recurrence.
Let us solve this by iterative approach,
$T(n) = T(n/2) + 1$ ..... (I)
put $n = n/2$ in equation (I) to find $T(n/2)$
$\Rightarrow T(n/2) = T(n/4) + 1$ .... (II)
substitute value of $T(n/2)$ in equation (I)
$\Rightarrow T(n) = (T(n/4) + 1) + 1$
$\Rightarrow T(n/2^2) + 2$ .....(III)
put $n = n/2$ in equation (II) to find $T(n/4)$
$\Rightarrow T(n/2/2) = T(n/2/4) + 2 + 1$
$\Rightarrow T(n/4) = T(n/8) + 1$
substitute value of $T(n/4)$ in equation (III),
$T(n) = T(n/8) + 3$

.
.
.

after k iterations
$T(n) = T(n/2^k) + k$
suppose $n = 2^k$,
$\Rightarrow \log n = \log 2^k$ ..... by taking log on both sides
$\Rightarrow k = \log_2 n$
$T(2^k) = T(2^k/2^k) + k \Rightarrow T(1) + k$
$\Rightarrow T(1) = 1$
So, $T(n) = 1 + k \Rightarrow T(n) = 1 + \log_2 n \Rightarrow$ **$T(n) = O(\log_2 n)$**

SORTING ALGORITHMS:

SELECTION SORT:

- Selection sort is a sorting algorithm, specifically an **in-place comparison sort.**
- It has **$O(n^2)$** time complexity, making it **inefficient** on large lists, and generally performs worse than the similar insertion sort.
- Selection sort is noted for its **simplicity**, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

- The algorithm divides the input list into two parts: the sub list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub list of items remaining to be sorted that occupy the rest of the list.
- Initially the sorted sub list is empty and the unsorted sub list is the entire input list.
- The algorithm proceeds by finding the smallest (or largest depending on sorting order) element in the unsorted sub list, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sub list boundaries one element to right.

Time Complexity:

- Worst Case Time Complexity: $O(n^2)$
- Average Case Time Complexity: $O(n^2)$
- Best Case Time Complexity: $O(n^2)$

BUBBLE SORT:

- Bubble sort, sometimes referred to as **Sinking Sort.**
- It is a simple sorting algorithm that repeatedly steps through the list to be sorted, **compares each pair of adjacent items and swaps them if they are in the wrong order.** The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.
- Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.

Complexity:

- Worst Case Time Complexity: O(n2)
- Average Case Time Complexity: O(n2)
- Best Case Time Complexity: O(n)

INSERTION SORT:

- Insertion Sort is a **simple** sorting algorithm, that builds the final sorted array (or list) one item at a time.
- When people manually sort cards in a bridge hand, most use a method that is similar to insertion sort.
- It is **much less efficient on large lists** than more advanced algorithms such as quick sort, merge sort and heap sort.

Advantages:

- simple implementation
- **efficient for (quite) small data sets**, much like other quadratic sorting algorithms.
- More efficient in practice than most other simple quadratic (i.e. $O(n^2)$) algorithms such as selection sort or bubble sort.
- **Adaptive:** i.e. efficient for data sets that are already substantially sorted.

- **Stable:** i.e. does not change the relative order of elements with equal keys
- **In-place:** i.e. only requires constant amount **O(1)** of additional memory space.
- **Online:** i.e. can sort a list as it receives it.

Best Case Time Complexity: O(n)
The best case input is an array that is already sorted.
Worst Case Time Complexity: $O(n^2)$
The simplest worst case input is an array sorted in reverse order
Average Case Time Complexity: $O(n^2)$
However, insertion sort is one of the fastest algorithm for sorting very small arrays, even faster than quick sort; indeed good quick sort implementations uses insertion sort for arrays smaller than certain threshold

QUICK SORT:

- Quick Sort is **divide-and-Conquer** algorithm.
- Quick Sort first divides a large array into two smaller sub arrays: the low elements and high elements.
- Quick Sort can then recursively sort sub-arrays.

Steps are:
1. Pick an element, called **pivot**, from the array
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater that the pivot come after it. (equal value can go either way). After this partitioning, the pivot is in its final position. This is called a partitioning operation.
3. Recursively apply the above steps to the sub array of elements with smaller value and separately to the sub-array of elements with greater values.

- **The base case of recursion is array of size zero or one,** which never need to sorted.
- The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.
- There are many different versions of quick sort that pick pivot in different ways.
    1. always pick first element as a pivot
    2. always pick last element as a pivot
    3. pick a random element as pivot
    4. pick median as a pivot

```
Algorithm QuickSort(int A[],int left,int right)
{
   int i = left;
   int j = right;

   //recursive calling termination condition
   if (left >= right)
        return;
   while( i < j )
   {
        While( i <= right && A[i] <= A[left] )
```

```
            i++;
        while( A[j] > A[left])
            j--;
        //if i and j are not crossed then swap them
        If( i < j )
            SWAP(A[i],A[j]);
    }//end of while loop

    //swap the pivot element with jth pos element
    SWAP(A[left], A[j]);
    //apply quick sort on left sub array
    QuickSort(A,left,j-1);
    //apply quick sort on right sub array
    QuickSort(A,j+1,right);
}//end of algorithm
```

Analysis of Quick Sort:

- Time taken by Quick Sort in general can be written as following:

$T(n) = T(k) + T(n-k-1) + \Theta(n)$.

- The first two terms are for two recursive calls, the last term is for the partition process, k is the number of elements which are smaller than pivot.
- the time taken by Quick Sort depends upon **the input array** and **partition strategy.**

Worst Case: $O(n^2)$

- **The worst case occurs when the partition process always picks greatest or smallest element as pivot.** If we consider partition strategy where first element always picked as a pivot, the worst case would occur when the array is already sorted in ascending or descending order.
- Following is the recurrence for worst case:

  $\Rightarrow T(n) = T(0) + T(n-1) + \Theta(n)$

  $\Rightarrow T(n) = T(n-1) + \Theta(n)$

the solution of above recurrence is **$\Theta(n^2)$**

Best Case: $\Theta(n \log n)$

- **the best case occurs when the partition process always picks the middle element as a pivot.**
- Following is the recurrence for the best case

  $T(n) = 2T(n/2) + \Theta(n)$
- the solution of above recurrence is **$\Theta(n \log n)$,** can be solved by using master theorem.

Average Case: $O(n \log n)$

- To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

- We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case:

T(n) = T(n/9) + T(9n/10) + Θ(n)
solution of above recurrence is also **O(n Log n)**.

- Although the worst case time complexity of Quick Sort is **$O(n^2)$** which is more than many other sorting algorithms like Merge Sort and Heap Sort, **Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real world data.**
- Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.
- However, **Merge Sort is generally considered better when data is huge and stored in external storage.**

Stack (ADT):

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data Organization In Memory:
  - Using Arrays:
    - int arr[MAX];
    - int top;
  - Using Linked List;
    - node *head;
    - Push() ==>AddFirst();
    - Pop() ==>DelFirst();
    - Peek() ==> return head->data;
    - IsEmpty() ==> return head==NULL;

Queue (ADT)

- Operations:
  - Push()
  - Pop()
  - Peek()
  - IsEmpty()
  - IsFull()
- Data organization in memory:
  - Using linked list (with head & tail pointer):
    - front => node *head;
    - rear => node *tail;
    - Push() =>AddLast()
    - Pop() =>DelFirst()
    - Peek() => return head->data;

- IsEmpty() => return head==NULL;

- Using arrays:
  - int arr[MAX];
  - int front;
  - int rear;

Linear Queue:

```
1. Init()
   front = rear = -1;
2. Push()
   rear ++;
   arr[rear] = ele;
3. Pop()
   front ++;
4. Peek()
   return arr[front+1];
5. IsEmpty()
   front == rear
6. IsFull()
   rear==MAX-1
```

Circular Queue:

- In linear queue, when rear reaches MAX-1, even if few locations are vacant, it shows queue is full. In other words, in linear queue there is no proper utilization of memory.
- To overcome this limitation, circular queue is implemented.
- In circular queue, front & rear are incremented in circular fashion i.e. 0, 1, 2, ..., MAX-1, 0, 1, ...

Implementation 1

```
1. Init()
   front = rear = -1;
2. Push()
   rear = (rear + 1) % MAX;
   arr[rear] = ele;
3. Pop()
   front = (front + 1) % MAX;
   if(front==rear)
        front = rear = -1;
4. Peek()
   temp = (front + 1) % MAX;
   return arr[temp];
5. IsEmpty()
   front == rear && front == -1
6. IsFull()
   (front==-1 && rear==MAX-1) || (front==rear && front!=-1)
```

Implementation 2 - Using count

```
1. Init()
   front = rear = -1;
   count = 0;
2. Push()
   rear = (rear + 1) % MAX;
   arr[rear] = ele;
   count++;
3. Pop()
   front = (front + 1) % MAX;
   count--;
4. Peek()
   temp = (front + 1) % MAX;
   return arr[temp];
5. IsEmpty()
   count == 0
6. IsFull()
   count == MAX
```

Types of Queues:

1. Linear Queue:
   a. No proper utilization of memory.
2. Circular Queue:
   a. Proper utilization of memory.
   b. Increments rear and front in circular fashion i.e. 0, 1, 2, 3, ..., MAX-1, 0, 1, ...
3. Priority Queue:
   a. Each element is associated with some priority.
   b. Element with highest priority is popped out first. (No FIFO behavior).
   c. Typically it is implemented as sorted linked list. While insertion element is added at appropriate position as per its priority. So insertion time complexity will be O(n). While deleting first (head) element is deleted (as it is element with highest priority).
4. Double Ended Queue (deque):
   a. Can push/pop elements from both sides i.e. front as well as rear.
   b. Usually implemented as doubly linked list with head & tail.

Stack Vs Queue:

	Stack	Queue
1	Stack is LIFO utility data structure.	Queue is FIFO utility data structure.
2	Push and Pop operations are done from the same end i.e. "top".	Push and Pop operations are done from different ends i.e. "rear" and "front" respectively.
3	There are no types of stack. However in single array you can implement two or more stacks (called as multi-stack approach).	There are four types of queues i.e. linear, circular, priority and deque.
4	Applications: • Function activation records are created on the stack	Applications: • Printer maintains queue of documents to be

for each function call. • To solve infix expression by converting to prefix or postfix. • Parenthesis balancing • To implement algos like Depth First Search.	printed. • OS uses queues for many functionalities: Ready queue, Waiting queue, Message queue. • To implement algos like Breadth First Search.

STL: Standard Template Library:

```
#include <stack>
stack<char> s;
s.push('A');
s.pop();
ele = s.top(); // s.peek()
s.empty();     // s.isempty()
```

```
#include <queue>
queue<char> q;
q.push('A');
q.pop();
ele = q.front(); // q.peek()
q.empty();       // q.isempty()
```

Infix, Prefix, Postfix:

- These are notations to represent math equations.
  - Infix: A + B -> For human
  - Prefix: + A B -> For computers
  - Postfix: A B + -> For computers
- Infix expression:
  - 4 + 3 - (9 - 5) / 2 * 8 - 6
- To convert Infix to Prefix/Postfix considers the priorities:
  1. ( )
  2. $ or ^ i.e. Exponential Operator
  3. * / % (left to right)
  4. + - (left to right)

Infix to Postfix using stack:

- Traverse Infix string element by element from left to right.
- If operand is found, then append to the postfix string.
- If operator is found, then push it on the stack.
- If priority of topmost of stack is greater or equal to priority of current operator, pop it from the stack and append to postfix string. Repeat this step if required.
- When infix string is completed, pop all operators from the stack and append to postfix string (one by one).
- If opening '(' is found, then push it on the stack.
- If closing ')' is found, pop all operators from the stack and append to postfix string (one by one) until '(' found on stack. Also pop and discard that '('.
- Example: 4 + 3 - (9 - 5) / 2 * 8 - 6

Symbol	Postfix	Stack
4	4	
+	4	+
3	4 3	+

-	4 3 +	-
(	4 3 +	- (
9	4 3 + 9	- (
-	4 3 + 9	- (-
5	4 3 + 9 5	- (-
)	4 3 + 9 5 -	-
/	4 3 + 9 5 -	- /
2	4 3 + 9 5 - 2	- /
*	4 3 + 9 5 - 2 /	- *
8	4 3 + 9 5 - 2 / 8	- *
-	4 3 + 9 5 - 2 / 8 * -	-
6	4 3 + 9 5 - 2 / 8 * - 6	-
	4 3 + 9 5 - 2 / 8 * - 6 -	

Infix to Prefix using stack:

1. Traverse Infix string element by element from right to left.
2. If operand is found, then append to the prefix string.
3. If operator is found, then push it on the stack.
4. If priority of topmost of stack is greater than priority of current operator, pop it from the stack and append to prefix string. Repeat this step if required.
5. When infix string is completed, pop all operators from the stack and append to prefix string (one by one).
6. If closing ')' is found, then push it on the stack.
7. If opening '(' is found, pop all operators from the stack and append to prefix string (one by one) until ')' is found on stack. Also pop and discard that ')'.
8. Reverse prefix string.

Postfix Evaluation:

1. Traverse postfix string from left to right.
2. If operand is found, push it on the stack.
3. If operator is found, pop two operands from the stack; calculate result and push it on stack.
   a. First popped is second operand; while second popped is first operand.
4. Repeat steps 1-2, until postfix string is completed.
5. At the end, pop final result from the stack.
6. Example: 4 3 + 9 5 - 2 / 8 * - 6 -

Symbol	Stack	Operation
4	4	
3	4 3	
+	7	4 + 3 = 7
9	7 9	
5	7 9 5	
-	7 4	9 - 5 = 4
2	7 4 2	
/	7 2	4 / 2 = 2
8	7 2 8	
*	7 16	2 * 8 = 16
-	-9	7 - 16 = -9

6	-9 6	
-	-15	-9 - 6 = -15

Prefix Evaluation:

1. Traverse prefix string from right to left.
2. If operand is found, push it on the stack.
3. If operator is found, pop two operands from the stack; calculate result and push it on stack.
4. First popped is first operand; while second popped is second operand.
5. Repeat steps 1-2, until prefix string is completed.
6. At the end, pop final result from the stack.

Linked List:

- **Terminologies:**
  - Linked list is a linear data structure that contains multiple records linked to each other.
  - Each item in the list is called as "Node".
  - Each node contains data and pointer (address) to the next node.
  - Typically linked lists are implemented as self-referential structure/class. The structure/class contains pointer of the same type to hold address of next node.
- **LinkedList (ADT):**
  - The commonly supported operations in a linked list is called as:
    A. AddFirst()
    B. AddLast()
    C. InsertAtPos()
    D. Search()
    E. DeleteFirst()
    F. DeleteLast()
    G. DeleteAtPos()
    H. Clear()
    I. Sort()
    J. Reverse()
  - **Linked list in C++:**

```
class list;
class node
{
  private:
      int data;
      node *next;
  public:
      node(intval=0);
      friend class list;
};
class list
{
private:
      node *next;
   public:
      list();
      ~list();
      addfirst(intval);
```

```
        addlast(intval);
        insert(intval, intpos);
        void delfirst();
        void dellast();
        void del(intpos);
        node* search(int key);
        void sort();
        void reverse();
        void clear();
};
```

- **Types of Linked Lists:**
  - **Singly Linear Linked List**
    - **A. Singly Linked List – only head pointer:**
      1. **Add First:O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      //newnode's next --> head
      newnode->next = head;
      // head -->newnode
      head= newnode;
}
```

      2. **Add Last: O(n)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=NULL)
            trav = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
}
```

      3. **Insert at position: O(n)**

```
// if list is empty or first pos, newnode is
```

```
first node.
if(head==NULL || pos==1)
      addfirst(val);
else
{
      // create and initialize node
      newnode = new node(val);
      // traverse till pos-1 node
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // newnode's next -->trav's next
      newnode->next = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
}
```

4. **Delete First: O(1)**

```
if(head!=NULL)
{
      // keep address of first node in temp
pointer
      temp = head;
      // take head to next node.
      head = head->next;
      // delete temp node.
      delete temp;
}
```

5. **Delete At Position: O(n)**

```
// if node to be deleted is first or list is
empty
if(pos==1 || head==NULL)
      delfirst();
else
{
      // traverse till pos-1
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // temp should point to the node to be
deleted.
      temp = trav->next;
      // trav's next --> temp's next
      trav->next = temp->next;
      // delete temp node
      delete temp;
}
```

6. **Delete All: O(n)**

```
// delete all nodes one by one, until list
become empty
while(head!=NULL)
      delfirst();
```

7. **Traverse Linked List:**

```
// start traversing from first node
trav = head;
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
      trav = trav->next;
} // repeat until end of list is reached
```

B. **Singly Linked List –head and tail pointer:**

1. **Add First: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = tail =newnode;
else
{
      // newnode's next --> head
      newnode->next = head;
      // head -->newnode
      head = newnode;
}
```

2. **Add Last: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = tail = newnode;
else
{
      // tail's next -->newnode
      tail->next = newnode;
      // tail -->newnode
      tail = newnode;
}
```

3. **Delete First: O(1)**

```
if(head!=NULL)
{
      // keep address of first node in temp
pointer
      temp = head;
      // take head to next node.
      head = head->next;
      // if deleted node was last node, then
      if(head==NULL)
            tail = NULL;
      // delete temp node.
      delete temp;
}
```

- **Singly Circular Linked List:**

  Last node's next pointer keeps address of first (head) node.

1. **Add First: O(n)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
      head = newnode;
      newnode->next = newnode;
}
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=head)
            trav = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
      // newnode's next --> head
      newnode->next = head;
      // head -->newnode
      head = newnode;
}
```

2. **Add Last: O(n)**

```
// if list is empty, newnode is first node.
if(head==NULL)
{
      head = newnode;
      newnode->next = newnode;
```

```
}
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=head)
            trav = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
      // newnode's next --> head
      newnode->next = head;
}
```

3. **Insert at position: O(n)**

```
// if list is empty or first pos, newnode is first
node.
if(head==NULL || pos==1)
      addfirst(val);
else
{
      // create and initialize node
      newnode = new node(val);
      // traverse till pos-1 node
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // newnode's next -->trav's next
      newnode->next = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
}
```

4. **Delete First: O(n)**

```
// if head is null, return
if(head==NULL)
      return;
// if there is a single node, delete it
else if(head==head->next)
{
      delete head;
      head = NULL;
}
// if there are more than one node
else
{
      // keep address of head node in temp pointer
      temp = head;
      // traverse till last node
```

```
        trav = head;
        while(trav->next!=head)
                trav = trav->next;
        // take head to next node.
        head = head->next;
        // update last node's next -->new head
        trav->next = head;
        // delete temp node.
        delete temp;
}
```

5. **Delete At Position: O(n)**

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
        delfirst();
else
{
        // traverse till pos-1
        trav = head;
        for(i=1; i<pos-1; i++)
                trav = trav->next;
        // temp should point to the node to be
deleted.
        temp = trav->next;
        // trav's next --> temp's next
        trav->next = temp->next;
        // delete temp node
        delete temp;
}
```

6. **Traverse Linked List:**

```
// start traversing from first node
trav = head;
do
{
        // visit current node
        printf("%d, ", trav->data);
        // go to the next node
        trav = trav->next;
} while(trav!=head);
// repeat until end of list is reached
```

- **Doubly Linear Linked List**

  Each node contains data, address of next node and address of previous node.

  1. **Add First: O(1)**

```
// create and initialize node
newnode = new node(val);
```

```
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // newnode's next --> head
      newnode->next = head;
      // cur head's prev -->newnode
      head->prev = newnode;
      // head -->newnode
      head = newnode;
}
```

2. **Add Last: O(n)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
      head = newnode;
else
{
      // traverse till last node
      trav = head;
      while(trav->next!=NULL)
            trav = trav->next;
      // trav's next -->newnode
      trav->next = newnode;
      // newnode'sprev --> last node (trav)
      newnode->prev = trav;
}
```

3. **Insert at position: O(n)**

```
// if list is empty or first pos, newnode is first
node.
if(head==NULL || pos==1)
      addfirst(val);
else
{
      // create and initialize node
      newnode = new node(val);
      // traverse till pos-1 node
      trav = head;
      for(i=1; i<pos-1; i++)
            trav = trav->next;
      // newnode's next -->trav's next
      newnode->next = trav->next;
      // newnode'sprev -->trav
      newnode->prev = trav;
```

```
        // next node's prev -->newnode
        if(trav->next!=NULL)
                trav->next->prev = newnode;
        // trav's next -->newnode
        trav->next = newnode;
}
```

4. **Delete First: O(1)**

```
if(head!=NULL)
{
        // keep address of first node in temp pointer
        temp = head;
        // take head to next node.
        head = head->next;
        // delete temp node.
        delete temp;
        // if node deleted is not last, make its prev
NULL
        if(head!=NULL)
                head->prev = NULL;
}
```

5. **Delete At Position: O(n)**

```
// if node to be deleted is first or list is empty
if(pos==1 || head==NULL)
        delfirst();
else
{
        // traverse till pos
        trav = head;
        for(i=1; i<pos; i++)
                trav = trav->next;
        // trav'sprev node's next -->trav's next
        trav->prev->next = trav->next;
        // trav's next node's prev -->trav'sprev
        if(trav->next!=NULL)
                trav->next->prev = trav->prev;
        // delete trav node
        delete trav;
}
```

6. **Delete All: O(n)**

```
// delete all nodes one by one, until list become
empty
while(head!=NULL)
        delfirst();
```

7. **Traverse Linked List:**

```
// start traversing from first node
trav = head;
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the next node
      trav = trav->next;
} // repeat until end of list is reached
```

8. **Traverse Linked List – In Reverse Direction:**

```
// traverse till last node
trav = head;
while(trav->next!=NULL)
      trav = trav->next;
// start traversing from last node
while(trav!=NULL)
{
      // visit current node
      printf("%d, ", trav->data);
      // go to the prev node
      trav = trav->prev;
} // repeat until beginning of list is reached
```

- **Doubly Circular Linked List**

  Each node contains data, address of next node and address of previous node. Last node's next contains address of first node, while first node's prev contains address of last node.

1. **Add First: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
      head = newnode;
      newnode->next = newnode;
      newnode->prev = newnode;
}
else
{
      // jump to the last node
      tail = head->prev;
      // newnode's next --> head
      newnode->next = head;
      // newnode'sprev -->tail
      newnode->prev = tail;
```

```
        // tail's next -->newnode
        tail->next = newnode;
        // cur head's prev -->newnode
        head->prev = newnode;
        // head -->newnode
        head = newnode;
}
```

2. **Add Last: O(1)**

```
// create and initialize node
newnode = new node(val);
// if list is empty, newnode is first node.
if(head==NULL)
{
        head = newnode;
        newnode->next = newnode;
        newnode->prev = newnode;
}
else
{
        // jump to the last node
        tail = head->prev;
        // newnode's next --> head
        newnode->next = head;
        // newnode'sprev --> tail
        newnode->prev = tail;
        // tail's next -->newnode
        tail->next = newnode;
        // cur head's prev -->newnode
        head->prev = newnode;
}
```

	Array	Linked list
1.	Array cannot grow or shrink dynamically (realloc() is not efficient).	Linked list can grow/shrink dynamically.
2.	Array has contiguous memory. Hence random access is possible.	Nodes are not in contiguous memory. Hence only sequential access is allowed.
3.	Arrays do not have memory overheads.	Linked lists have memory overheads e.g. each node contains address of 'next' node.
4.	Insert/Delete at middle position need to shift remaining elements. Hence will be slower.	Insert/Delete at middle position need to modify links (next/prev pointers). Hence will be faster.
5.	To deal to fixed number of elements and frequent random access, arrays are better.	To deal to dynamic number of elements and frequent insertion/deletion operators, linked lists are better.

	Singly Linked List with Head	Singly Linked List with Head & Tail
1.	Add last operation is slower i.e. O(n)	Add last operation is faster i.e. O(1)
2.	No overload.	Extra overhead i.e. tail pointer.
3.	Simplified list manipulation operations.	List manipulation need to consider tail pointer wherever appropriate.

	Singly Linked List	Doubly Linked List
1.	Unidirectional access.	Bi-directional access.
2.	For search and delete operation two pointers are required (Need to maintain prev pointer) during traversing.	For search and delete operation only one pointer is required (No need to maintain prev pointer) during traversing.
3.	Simplified list manipulation operations.	List manipulation need to consider prev pointer of each node in all operations.

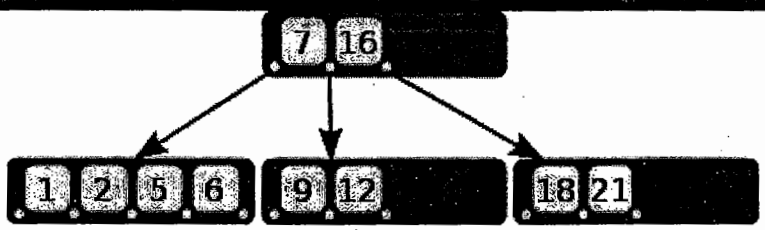
	Doubly Linked List	Doubly Circular Linked List
1.	Traversing till last node is slower (add last operation) i.e. O(n)	Traversing till last node is faster (add last operations) i.e. O(1)

Trees

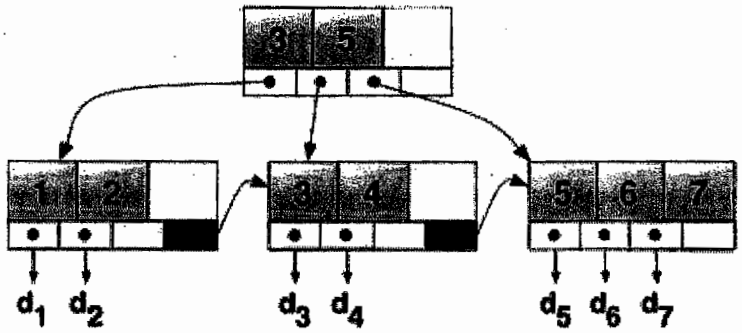
- **Terminologies:**
  - Tree is a non-linear data structure in which one specially designated node is called as "root" (i.e. starting point of the tree) and remaining elements can be partitioned into "m" disjoint subsets so that each subset is a tree itself.
  - Number of child nodes for any node is called as "degree of a node".
  - Maximum degree of a child in tree is called as "degree of the tree".
  - All the nodes in the path from the root to that node, are called "ancestors" of that node.
  - All the nodes accessible from the given node, are called as "descendants" of that node.
  - Child nodes of the same parent are called as "Siblings".
  - Terminal node of the tree is called as "leaf node". Leaf node does not have child nodes.
  - Level of node:
    - Indicates position of the node in tree hierarchy.
    - Level of child = Level of parent + 1
    - Level of root = 0
  - The maximum level of a node is "Height of tree".
  - Tree with no nodes (i.e. empty tree) is called as "Null Tree". Height of Null tree is -1.
- **Types of Trees:**
  - **Binary Tree:**
    - Tree in which each node has maximum two child nodes, is called as "binary tree".
    - Binary tree has degree 2. Hence it is also called as 2-tree.
  - **Binary Search Tree:**
    - The binary tree in which left child node is always smaller and right child node is always greater or equal to the (parent) node; is called as "Binary Search Tree".
    - Searching is faster. Time complexity:O(h). Where "h" is height of the tree.
  - **Complete Binary Tree:**
    - Binary tree in which all leaf nodes are at the same level.
  - **Strictly Binary Tree:**
    - Binary tree in which each non-leaf node has exact two child nodes.
  - **Full Binary Tree:**
    - Binary tree with its full capacity for the given height.
    - In other words, adding one more node will increase height of the tree.
    - It is always complete as well as strictly binary tree.
    - Number of elements = $2^{(h+1)} - 1$
  - **Almost complete binary tree:**
    - The binary tree which follows two conditions:
      - All leaf nodes are at level h or h-1.
      - All leaf nodes at last level (h), are aligned to left as much as possible.
  - **Threaded Binary Tree:**

- These trees will allow very fast in-order traversal (ascending or descending).
- Binary tree in which right null link of each node is replaced by address of its in-order successor; such tree is called as "right threaded binary tree".
- Binary tree in which left null link of each node is replaced by address of its in-order predecessor; such tree is called as "left threaded binary tree".
- If right as well as left null links of each node is replaced by its in-order successor and in-order predecessor respectively; then such tree is called as "in threaded binary tree".

- **Skewed Binary Tree:**
  - The binary tree in which only left link is used to keep address of child node (right link of each node is kept null), is called as "left skewed binary tree".
  - The binary tree in which only right link is used to keep address of child node (left link of each node is kept null), is called as "right skewed binary tree".
  - Height of skewed binary tree is maximum for given number of nodes.Hence searching will be too slow i.e. O(n)
- **Balanced binary tree:**
  - In binary search trees, searching is faster if height of the tree is kept as minimum as possible.
  - The binary search tree with minimum possible height (for given number of nodes) is called as "balanced binary search tree" or "height balanced binary search tree".
  - Balance factor of node = height of left sub tree - height of right sub tree.
  - The binary search tree, in which balance factor of each node is in range of -1 to 1, is called as "balanced binary search tree".
  - On any binary search tree, series ofappropriate left rotations and right rotations on appropriate nodes can convert it into "balanced binary search tree".
    - If balance factor of a node < -1, apply left rotation on that node.
    - If balance factor of a node > +1, apply right rotation on that node.
- **AVL Tree:**
  - Invented by "GreogyAdelson-Velsky" and "E.M. Landis".
  - Self-balancing binary search tree.
  - After each insert or delete operation, appropriate rotations are applied to make tree balanced again.
- **B Tree:**
  - The B-tree is a generalization of a binary search tree in that a node can have more than two children.
  - In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range.
  - In order to maintain the pre-defined range, internal nodes may be joined or split.
  - Each internal node of a B-tree will contain a number of keys. The keys act as separation values which divide its sub-trees.
  - A B-tree is kept balanced by requiring that all leaf nodes be at the same depth (level).



- **B+ Tree:**
  - Like B trees but with few changes as follows:
  - In the B+ tree, copies of the keys are stored in the internal nodes.
  - The keys and records are stored in leaves.
  - A leaf node may include a pointer to the next leaf node to speed sequential access.



Binary Search Tree:

- Implementation(Design) of binary search tree:

```
class tree;
class node
{
private:
   int data;
   node *left, *right;
public:
   node(intval=0);
   friend class tree;
};
class tree
{
private:
   node *root;
public:
   tree();
   void add(intval); // add node
   node* search(int key); // search node
   node* search(int key, node **parent);// search node with
parent
   void preorder(node *trav); // preorder traversal
```

```
        void preorder(); // wrapper for preorder(root);
        void inorder(node *trav); // inorder traversal
        void inorder(); // wrapper for inorder(root);
        void postorder(node *trav); // postorder traversal
        void postorder();     // wrapper for postorder(root);
        void height(node *trav); // height of tree
        void height();  // wrapper for height(root);
        void clear(node *trav); // delete node and its
descendants
        void clear();    // wrapper for clear(root);
        void preorder_nonrec(); // preorder traversal non-
recursive
        void inorder_nonrec(); // inorder traversal non-recursive
        void postorder_nonrec(); // postorder traversal non-
recursive
        void del(int key); // search and delete node
    };
```

- Add a node in tree:

```
void add(intval)
{
        // create and initialize node
        node *newnode = new node(val);
        // if tree is empty
        if (root == NULL)
                root = newnode;
        else
        {
                node *trav = root;
                while (true)
                {
                        if (val<trav->data)
                        {
                                if (trav->left == NULL)
                                {
                                        trav->left = newnode;
                                        break;
                                }
                                trav = trav->left;
                        }
                        else
                        {
                                if (trav->right == NULL)
                                {
                                        trav->right = newnode;
                                        break;
```

```
                        }
                        trav = trav->right;
                    }
                }
            }
}
```

- **Tree Traversal:**
  - Pre-order: Parent , Left, Right
  - In-order: Left, Parent, Right
  - Post-order: Left, Right, Parent

```
void preorder(node *trav)
{
      if (trav == NULL)
            return;
      printf("%d, ", trav->data);
      preorder(trav->left);
      preorder(trav->right);
}
void inorder(node *trav)
{
      if (trav == NULL)
            return;
      inorder(trav->left);
      printf("%d, ", trav->data);
      inorder(trav->right);
}
void postorder(node *trav)
{
      if (trav == NULL)
            return;
      postorder(trav->left);
      postorder(trav->right);
      printf("%d, ", trav->data);
}
```

- **Delete All Nodes in Tree:**

```
void clear(node *trav)
{
      if (trav == NULL)
            return;
      clear(trav->left);
      clear(trav->right);
      delete trav;
}
void clear()
{
```

```
        clear(root);
        root = NULL;
}
```

- **Height of tree:**

```
int height(node *trav)
{
        if (trav == NULL)
                return -1;
        int hl = height(trav->left);
        inthr = height(trav->right);
        int max = (hl >hr ? hl : hr);
        return max + 1;
}
int height()
{
        return height(root);
}
```

- **Search a node with its parent:**

```
node* search(int key, node **parent)
{
        node *trav = root;
        *parent = NULL;
        while (trav != NULL)
        {
                if (key == trav->data)
                        return trav;
                *parent = trav;
                if (key <trav->data)
                        trav = trav->left;
                else
                        trav = trav->right;
        }
        *parent = NULL;
        return NULL;
}
```

- **Delete a node:**
  - if node's left is null:

```
if(temp->left==NULL)
{
        if(temp==root)
                root = temp->right;
        else if(temp==parent->left)
                parent->left = temp->right;
        else
```

```
            parent->right = temp->right;
}
```

- if node's right is null:

```
if(temp->right==NULL)
{
      if(temp==root)
            root = temp->left;
      else if(temp==parent->left)
            parent->left = temp->left;
      else
            parent->right = temp->left;
}
```

- if node's left as well as right is NOT null:

```
if(temp->left!=NULL && temp->right!=NULL)
{
      // find the succ of temp along with its (succ)
parent
      parent = temp;
      succ = temp->right;
      while(succ->left!=NULL)
      {
            parent = succ;
            succ = succ->left;
      }
      // replace temp data by succ data
                  temp->data = succ->data;
      // mark succ for deletion
      temp = succ;
}
```

- deleting node (full) - steps:
  - Search node to be deleted along with its parent.
  - If node's left as well as right is NOT null, deletion code for that.
  - If node's left is null, deletion code for that.
  - Else if node's right is null, deletion code for that.
  - delete memory of temp node.

- **Pre-order non-recursive:**

```
void preorder_nonrec()
{
      printf("NONREC PRE : ");
      node *trav = root;
      stack<node*> s;
      while (trav != NULL || !s.empty())
      {
            while (trav != NULL)
            {
                  printf("%d, ", trav->data);
                  if (trav->right != NULL)
```

```
                    s.push(trav->right);
                trav = trav->left;
            }
            if (!s.empty())
            {
                trav = s.top();
                s.pop();
            }
        }
        printf("\n");
}
```

- **In-order non-recursive:**

```
void inorder_nonrec()
{
    printf("NONREC IN  : ");
    node *trav = root;
    stack<node*> s;
    while (trav != NULL || !s.empty())
    {
        while (trav != NULL)
        {
            s.push(trav);
            trav = trav->left;
        }
        if (!s.empty())
        {
            trav = s.top();   s.pop();
            printf("%d, ", trav->data);
            trav = trav->right;
        }
    }
    printf("\n");
}
```

- **Post-order non-recursive:**

```
void postorder_nonrec()
{
    printf("NONREC POST: ");
    node *trav = root;
    stack<node*> s;
    while (trav != NULL || !s.empty())
    {
        while (trav != NULL)
        {
            s.push(trav);
            trav = trav->left;
        }
```

```
            if (!s.empty())
            {
                  trav = s.top(); s.pop();
                  if (trav->right == NULL || trav->right-
>visited == true)
                  {
                        printf("%d, ", trav->data);
                        trav->visited = true;
                        trav = NULL;
                  }
                  else
                  {
                        s.push(trav);
                        trav = trav->right;
                  }
            }
      }
      printf("\n");
}
```

- **Depth First Search:**

```
node* bfs(int key)
{
      node *trav;
      queue<node*> q;
      //push root on queue
      q.push(root);
      //repeat until queue is empty
      while (!q.empty())
      {
            // pop an ele from queue
            trav = q.front();     q.pop();
            // compare with ele to search
            if (key == trav->data)
                  return trav;
            // if trav has left child, push on queue
            if (trav->left != NULL)
                  q.push(trav->left);
            // if trav has right child, push on queue
            if (trav->right != NULL)
                  q.push(trav->right);
      }
      return NULL;
}
```

- **Breadth First Search:**

```
node* dfs(int key)
{
```

```
        node *trav;
        stack<node*> s;
        //push root on stack
        s.push(root);
        //repeat until stack is empty
        while (!s.empty())
        {
              // pop an ele from stack
              trav = s.top(); s.pop();
              // compare with ele to search
              if (key == trav->data)
                    return trav;
              // if trav has right child, push on stack
              if (trav->right != NULL)
                    s.push(trav->right);
              // if trav has left child, push on stack
              if (trav->left != NULL)
                    s.push(trav->left);
        }
        return NULL;
}
```

Array Implementation of tree:

- A binary tree can be simulated in an array i.e. parent-child relationship can be maintained.
- Typically array begin with index 1. Then
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1
- Array implementation of almost complete binary tree, is called as "Heap" data structure.
- If each parent element in heap is greater than both of its child elements, then it is called as "Max Heap".
- If each parent element in heap is less than both of its child elements, then it is called as "Min Heap".
- Heap sort algorithm uses Max heap (ascending sort) or Min heap (descending sort).

HASHING:

Suppose we want to design a system for **storing employee records keyed using phone numbers**, and we want following queries to be performed efficiently:

- **insert** a phone number and corresponding information
- **delete** a phone number and related information
- **search** a phone number and fetch the information

We can think of using the following data structures to maintain information about different phone numbers.

1. array: of phone numbers and records
2. linked list: of phone numbers and records
3. balanced binary search tree with phone numbers as keys
4. direct access table

- For **arrays and linked lists:** we need to search in a linear fashion which can be costly in practice. If we use array and keep the data stored, then a phone number can be searched in **O(log n)** time using binary search, but insert and delete operations becomes costly as we have to maintain sorted order.
- With **balanced binary search tree,** we get moderate search, insert and delete times. All of these operations can be guaranteed to be in **O(log n)** time.
- Another solution that one can think of is to use a **direct access table:** where we make a big array and use phone numbers as index in the array. An entry in an array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number.
- **Time complexity wise this solution is the best among all, we can do all operations in O(1) time.**
- For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.
- This solution has many practical limitations. First problem with this solution **extra space required is huge.**
- For example **if phone number is n digits, we need O(m*10^n)** space for table where **m is size of pointer to record.** Another problem is an integer in a programming language may not store n digits.
- Due to above limitations **Direct Access Table** cannot always be used.
- **Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Arrays, Linked List, Balanced BST in practice.**
- **With hashing we get O(1) search time on average(under reasonable assumptions) and O(n) in worst case.**
- **Space required is: average and worst case: O(n)**
- The **average case** and **worst case** time complexity of **search**, **insert** and **delete** operations takes **O(1)** time.
- **Hashing is an improvement on Direct Access Table.**
- **The idea is to use hash function that converts a given phone number or any other key into a smaller number and uses the small number as index in a table called hash table.**

- **Hash Table: Hash Table (hash map)** is a data structure which implements an **associative array abstract data type, a structure that can map keys to values.**

- **Associative array:** In computer science **associative array, map, symbol table**, or **dictionary** is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

  Operations associated with this data type as follows:
  1. the addition of pair of the collection
  2. the removal of a pair from the collection
  3. the modification of an existing pair
  4. the lookup of a value associated with a particular key

- In **associative array, the association between a key and value is often known as a binding,** and the same word binding may also used to refer to the process of creating a new association.

- The operations that are usually defined for an associative array are:
  1. Add or insert: add a new (key, value) pair to the collection. The arguments to this operation are the key and value.
  2. Reassign: replace the value in one of the (key, value) pairs that are already in the collection, binding an old key to a new value. As with an insertion, the argument to this operation is the key and the value.
  3. Remove or Delete: remove a (key, value) pair from the collection, unbinding a given key from its value. The argument to this operation is key.
  4. Lookup: find the value (if any) that is bound to a given key. The argument to this operation is the key, the value is returned from the operation. If no value is found, some associative array raise an exception

- A hash table uses **hash function** to compute an index into an array of **buckets** or **slots**, from which the desired value can be found.

- **Hash Function:**
- A Hash Function, **is any function that can be used to map data of arbitrary size to data of fixed size.**
- A function **that converts a given big phone number into a small practical integer value.** The mapped integer value is used as an index in the hash table.
- In simple terms, **a hash function maps a big number or string to a small integer that can be used as index in hash table.**
- The values returned by hash function are called as **hash values, hash codes, digests** or simply **hashes.**

- **Applications of hash table:**

- widely used in **computer software for rapid data lookup.** Hash functions **accelerates table or database lookup by detecting duplicated records in a large file.**
- They are also used in **Cryptography.**
- Hash functions are also used **to build caches for large data sets stored in slow media.**

- A good hash function should have following properties:
  1. Efficiently Computable
  2. Should uniformly distribute the keys(each table position equally likely for each key).

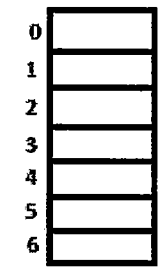
- For example for phone numbers, a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

- **Hash Table:**
- **An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number, has hash function value equal to the index for the entry.**
- In Computing, a **hash table (hash map) is a data structure which implements an associative array abstract data type, a structure that can map keys to values.**
- A hash table uses **hash function** to compute an index into an array of **buckets** or **slots**, from which the desired value can be found.
- Ideally, **the hash function will assign each key to a unique bucket/slot,** but most hash table designs employ an imperfect hash function, which might cause **hash collisions** where **the hash function generates the same index for more than one key.**
- **Hash functions** are used in **hash tables, to quickly locate a data record** (e.g. a **dictionary** definition) given its search key(the headword). Specifically, **the hash function is used to map the search key to an index;** the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, **are used to implement associative arrays and dynamic sets.**

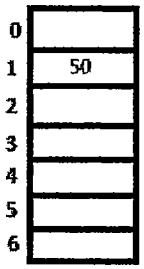
- **Collision Handling:**
- **Collision:** Since a hash function gives us a small number for a big key, there is possibility that two keys result in same value. **The situation where a newly inserted key maps to an already occupied slot in hash table is called collision** and must be handled using some collision handling technique.
- **Following are the ways to handle the collisions:**
  1) **Separate Chaining:** The idea is **to make each cell of hash table point to a linked list of records that have same hash function value.** Chaining is simple, but requires additional memory outside the table.
  2) **Open Addressing:** In open addressing, **all elements are stored in the hash table itself.** Each table entry contains either a record or NIL. When searching for an element, we one by one examine table's slots until the desired element is found or it is clear that the element is not in the table.

- **Separate Chaining:**
- **What is Collision?**
  Since a hash function gets us a small number for a key which is big integer or string, there is possibility that two keys result in same value. **The situation where newly inserted key maps to an already occupied slot in hash table is called collision** and must be handled using some collision handling technique.

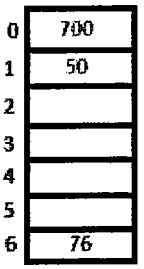
- The idea is **to make each cell of hash table point to a linked list of records that have same hash function value.**
- Let us consider a simple hash function as **"key mod 7"** and sequence keys as **50, 700, 76, 85, 92, 73, 101.**



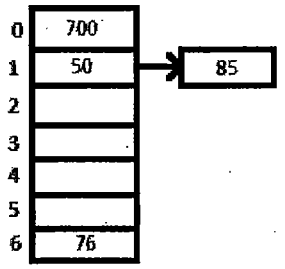
Initial Empty Table



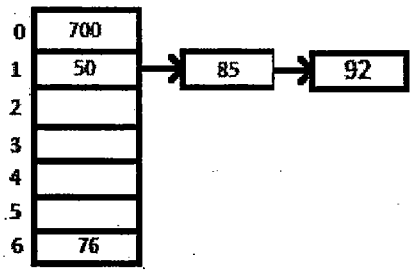
Insert 50



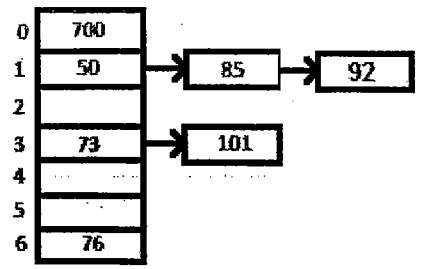
Insert 700 and 76



Insert 85: Collision Occurs, add to chain



Inser 92 Collision Occurs, add to chain



Insert 73 and 101

- **Advantages:**
  1. Simple to implement
  2. Hash table never fills up, we can always add more elements to chain
  3. Less sensitive to the hash function or load factors
  4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- **Disadvantages:**

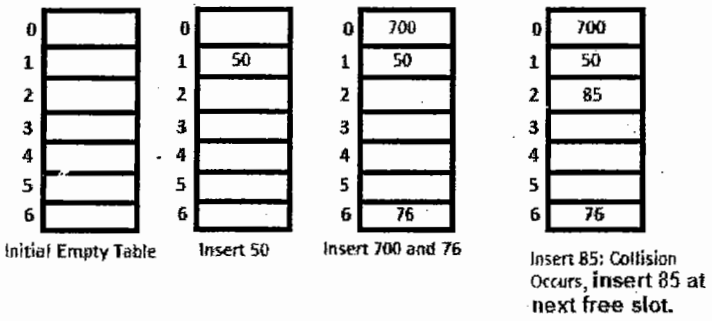
1. Cache performance of chaining is not good as keys are stored using linked list. **Open addressing provides better cache performance** as everything is stored in same table.
2. **Wastage of space** (some parts of hash table are never used)
3. **If the chain becomes long, then search time can become O(n) in worst case.**
4. **Uses extra space for links**

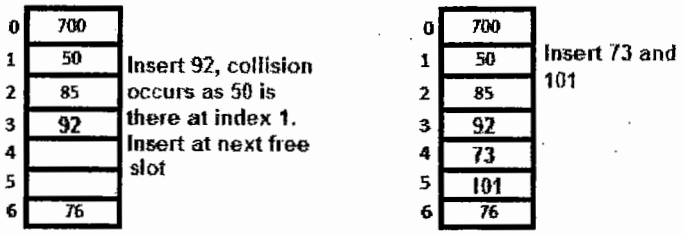
- **Performance of Chaining:**
  Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing)

  m = number of slots in hash table
  n = number of keys to be inserted in hash table
  Load Factor => LF = n / m
  expected time to search: **O(1 + LF)**
  expected time to insert/delete = **O(1 + LF)**
  Time complexity of search, insert and delete is: O(1)
  if LF is O(1).

- **Open Addressing:**
- Like Separate Chaining, Open Addressing is a method for handling collisions.
- In **Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (note that we can increase table size by copying old data if needed).**
- **Insert(k):Keep probing until an empty slot is not found. Once an empty slot is found, insert key.**
- **Search(k):Keep probing until slot's key doesn't become equal to k or an empty slot is reached.**
- **Delete(k):Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".**
- **Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.**
  [Probe: investigate, to search into, look into]
- **Open Addressing is done following ways:**
  a) **Linear Probing:** In Linear Probing, we linearly probe for the next slot. For example, typical gap between two probes is 1 as taken in below example also.
  Let **hash(x)** be **the slot index computed using hash function** and **S** be **the table size.**
  if slot **hash(x) % S** is full, then we try **(hash(x)+1)%S**
  if**(hash(x)+1)%S** is also full, then we try **(hash(x)+2)%S**
  if**(hash(x)+2)%S** is also full, then we try **(hash(x)+3)%S**
  .....
  .....

Let us consider a simple **hash function as "key mod 7"** and sequence of keys are: **50, 700, 76, 85, 92, 73, 101.**





- **Clustering:** The main problem with Linear Probing is **Clustring, many consecutive elements forms groups and it starts taking time to find a free slot or to search an element.**

b) **Quadratic Probing:** We look for $i^{2\,th}$ slot in i^{th} iteration
Let **hash(x)** be **the slot index computed using hash function** and **S be the table size.**

If slot **hash(x)%S** is full, then we try **(hash(x)+1*1)%S**
If **(hash(x)+1*1)%S** is also full, then we try **(hash(x)+2*2)%S**
If **(hash(x)+2*2)%S** is also full, then we try **(hash(x)+3*3)%S**
.....
.....

c) **Double Hashing:** We use another hash function **hash2(x)** and look for **i*hash2(x)** slot in i^{th} iteration.
Let **hash(x)**be the slot index computed using hash function and S be the table size.

If slot **hash(x)%S** is full, then we try **(hash(x)+1* hash2(x))%S**
If **(hash(x)+1*hash2(x))%S** is also full, then we try **(hash(x)+2*hash2(x))%S**
If **(hash(x)+2*hash2(x))%S** is also full, then we try **(hash(x)+3*hash2(x))%S**
.....
.....

- **Comparison of above three:**
- **Linear Probing** has the **best cache performance**, but suffers from **clustering.** One more advantage of Linear Probing is **easy to compute.**

- **Quadratic Probing** lies between the two in terms of cache performance and clustering.
- **Double Hashing** has **poor cache performance** but **no clustering.** Double Hashing **requires more computation time as two hash functions need to be computed.**

- **Open Addressing v/s Separate Chaining:**
- **Advantages of Chaining:**
  1. Chaining is **simpler to implement**
  2. In Chaining, **Hash table never fills up,** we can always add more elements to chain. In open addressing, **table may become full.**
  3. Chaining is **less sensitive to the hash function or load factor.**
  4. Chaining is **mostly used when it is unknown how many and how frequently keys may be inserted or deleted.**
  5. Open Addressing **requires extra care for to avoid clustering and load factor.**
- **Advantages of Open Addressing:**
  1. Cache Performance of chaining is not good, as keys are stored using linked list. **Open addressing provides better cache performance** as everything is stored in same table.
  2. **Wastage of Space** (Some parts of hash table in chaining are never used). In Open Addressing, a slot can be used even if an input doesn't map to it.
  3. Chaining **uses extra space for links.**

- **Performance of Open Addressing:**

Like Chaining, performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = no. of slots in hash table
n = no. of keys to be inserted in hash table

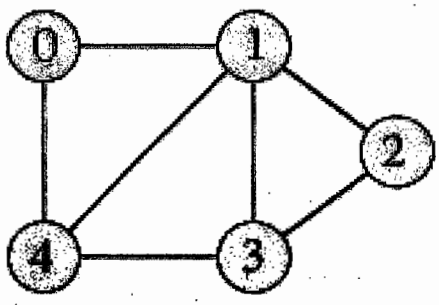
load factor LF = n / m (< 1)
expected time to search/insert/delete < 1/(1-LF)
so search, insert and delete takes: O(1/(1-LF)) time.

GRAPH:

- A graph is a **non-linear advanced data structure** that consists of two components:
  1. A finite set of **vertices** also called as **nodes**
  2. A finite set of ordered pair of the form (u, v) called as **edge** or **arc.**
  -> The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(**di-graph**).
  -> The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v.
  -> The edges may contain **weight/cost.**
  Different data structures for the representation of the graphs are used in practice:

- **Applications of Graph:**
- Graphs are used to represent many real life applications:
- Graphs are used to represent networks. The network may includes paths in a city or telephone network or circuit network.
- Graphs are also used in social network sites like LinkedIn, Facebook. For example in Facebook each person represented with a vertex (or a node). Each node is a structure and contains information like person_id, name, gender and locale.
- Graphs can be model many types of relations and processes in physical, biological, social and information systems. Many practical problems can be represented by graphs.

- **Representation of Graph:**
  Graph can be represented by two ways:
  1. Adjacency Matrix (using 2D array)
  2. Adjacency List (using linked list)



1. Adjacency Matrix (using 2D array):

- Adjacency matrix is 2D array of size V*V where V is the number of vertices in a graph.
- Let the 2D array be: adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric matrix.
- Adjacency matrix is also used to represent weighted graphs. If adj[i][j]=w, then there is an edge from vertex i to vertex j with weight w.
- Example: Un-directed Un-weighted Graph: G
  V = { 0,1,2,3,4 } and
  E = { (0,1), (0,4),(1,2), (1,3), (1,4), (2,3), (3,4) }.

MATRIX REPRESENTATION OF ABOVE GRAPH:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

PROS:

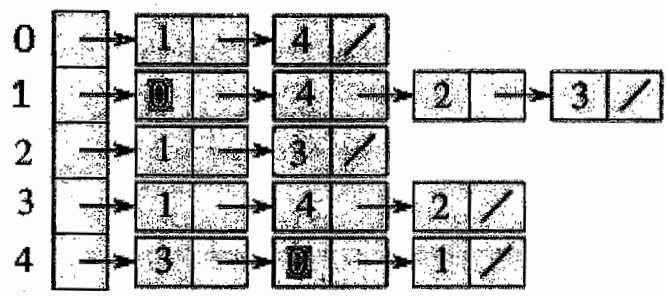
- Representation is easier to implement and follow.
- Removing an edge takes O(1) time.
- Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done in O(1).

CONS:

- Consumes more space $O(V^2)$. Even if the graph is sparse (contains less no. of edges), it consumes the same space
- Adding a vertex is $O(V^2)$ time.

2. Adjacency List (using linked list):

- Following is the **adjacency list representation** of the above graph.



- An array of linked list is used.
- Size of the array is equal to number of vertices
- Let the array be arr[]. An entry of arr[i] represents the linked list of vertices adjacent to the i^{th} vertex.
- This representation can also used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.

PROS:

- Saves space O(|V|+|E|).
- In the worst case, there can be C(V,2) number of edges in a graph, thus consuming $O(V^2)$ space.
- Adding vertex is easier.

CONS:

- Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

SPANNING TREE: is a sub-graph of a graph which remains connected after removing one/more edges from it and does not contains a cycle.

FOREST:

- A Forest is an undirected graph, all of whose components are trees.
- In another words, the graph consists of a disjoint union of trees.
- Equivalently, a forest is an undirected cycle free graph.

GRAPH TRAVERSAL ALGORITHMS:

BFS: BREADTH FIRST TRAVERSAL ALGORITHM

1. push any node (starting vertex) into the queue and mark it
2. pop vertex from the queue and visit it (display)
3. push its all unmarked adjacent vertices (neighbors) into the queue and mark them
4. repeat steps 2&3 until queue is empty.

DFS: DEPTH FIRST TRAVERSAL ALGORITHM

1. Push any node (starting vertex) into the stack and mark it
2. Pop vertex from the stack and visit it (display)
3. Push all its unmarked adjacent vertices (neighbors) into the stack and mark them
4. Repeat steps 2 & 3 until stack is empty.

Greedy Algorithms: (Activity Selection Problem)

- Greedy is an algorithm paradigm **that builds up a solution piece by piece, always choosing the next that offers the most obvious and immediate benefit.**
- Greedy algorithms are used for **optimization problems**.
- Optimization problems can be solved using Greedy if the problems have the following property: **at every step, we can make a choice that looks best at that moment, and we get the optimal solution of the complete problem.**
- If Greedy algorithms can solve a problem then generally it becomes the best method to solve that problem.
- The **Greedy algorithms** are in general **more efficient** than other techniques like **Dynamic Programming.**
- But Greedy algorithms cannot always be applied, for example, **Fractional Knapsack Problem** can be solved using Greedy, but **0-1 Knapsack** cannot be solved by using Greedy.
- Following are some standard algorithms that are Greedy Algorithms:
  1. **Dijkstra's Shortest Path:** Dijkstra's algorithm is very similar to Prim's algorithm. **The shortest path tree is built up, edge by edge.** We maintain two sets: set of vertices already included in tree and the set of vertices not yet included. The Greedy choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.
  2. **Kruskal's Minimum Spanning Tree (MST):** In Kruskal's Algorithm, **we create a MST by picking edges one by one.** The Greedy choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

3. **Prim's Minimum Spanning Tree (MST):** In Prim's algorithm **we create a MST by picking edges one by one.** We maintain two sets: set of vertices already included in MST and the set of vertices not included yet. **The Greedy choice is to pick the smallest weight edge that connects the two sets.**
4. **Huffman Coding: Huffman Coding is a loss-less compression technique.** It assigns variable length bit codes to different characters. The Greedy choice is to assign least bit length code to the most frequent character.

- Greedy Algorithms are sometimes also used **to get approximation for hard optimization problems.** For example, **Travelling Salesman Problem** is a **NP Hard Problem**. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. This solutions doesn't always produce the best optimal solution, but can be used to get an approximate optimal solution.

- **The Greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity.** We can sort the activities according to their finishing time so that **we always consider the next activity as minimum finishing time activity.**

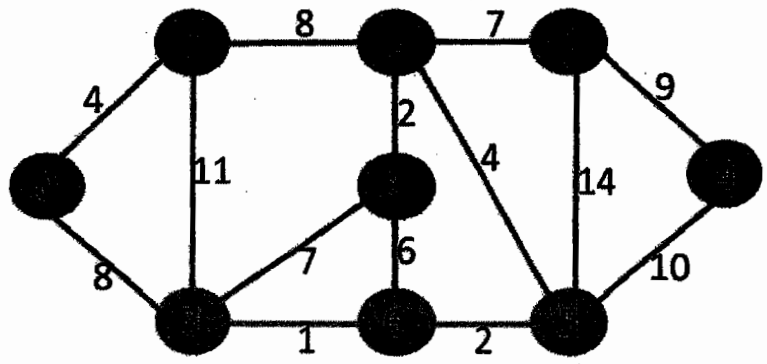
  1. Sort the activities according to their finishing time
  2. Select the first activity from the sorted array and print it
  3. Do the following for remaining activities in the sorted array.

- if the start time of this activity is greater than the finish time of previously selected activity then select this activity and print it.

Dijkstra's Algorithm (Shortest Path Algorithm):

- Given an undirected weighted input graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.
- Dijsktra's Algorithm is very similar to Prim's algorithm for Minimum Spanning Tree.
- Like Prim's MST, we generate a SPT(Shortest Path Tree) with given source as root.
- **We maintain two sets: one set contains vertices included in shortest path tree, and another set includes vertices not included in shortest path tree.**
- **At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.**

Example input graph:



- Time complexity of the implementation is $O(V^2)$, if the input graph is represented using adjacency list, it can be reduced to **O(E. log V)** with the help of binary heap.
- Dijkstra's algorithm doesn't work for graphs with negative weight edges. Bellman-Ford algorithm can be used for the same.

PRIM'S ALGORITHM:

- Algorithm to find **Minimum Spanning Tree (MST) for a weighted undirected** graph.
- This means **it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.**
- The algorithm operates by building this MST, one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.
- The algorithm was developed in **1930** by Czech mathematician **Vojtech Jarnik** and later rediscovered and republished by computer scientists **Robert C. Prim in 1957** and **Edsger W. Dijkstra in 1959.** Therefore it is also called as the **DJP algorithm, Jarnik's algorithm**, the **Prim-Jarnikalgorithm** or the **Prim's-Dijsktra algorithm.**

TIME COMPLEXITY:

- The time complexity of Prim's algorithm depends on the data structure used for the graph and for ordering the edges by weight, which can be done by using a priority queue.

Minimum edge weight data structure	Time Complexity(total)
Adjacency matrix, searching	$O(\|V^2\|)$
Binary heap and adjacency list	$O((\|V\|+\|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

- A simple implementation of Prim's, using an **adjacency matrix** or **adjacency list** graph representation and linearly searching an array of weights to find the minimum weight edge, to add requires $O(|V^2|)$ running time.
- However, this running time can be greatly improved further by using **heaps** to implement finding minimum weight edges in the algorithms inner loop.
- A first improved version uses a heap to store all edges on the input graph, ordered by their weight. This leads to an $O(|E| \log |E|)$ worst case running time. But storing vertices instead of edges can improve it still further. The heap should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree MST (or infinity if no such edge exists).
- Every time a vertex V is chosen and added to the MST, a decrease key operation is performed on all vertices w outside the partial MST such that v is connected to w, setting the key to the minimum of its previous value and the edge cost of (v, w).
- Using a simple **binary heap** data structure, Prim's algorithm can now be shown to run in $O(|E| \log |V|)$ where |E| is the number of edges and |V| is the number of vertices.
- Using a more sophisticated **Fibonacci heap**, this can be brought down to $O(|E|+|V| \log |V|)$, which is asymptotically faster when the graph is dense enough that |E| is $w(|V|)$ and linear time when |E| is at least $|V| \log |V|$. For graphs of even greater density (having at

least $|V|^c$ edges for some $c > 1$), Prim's algorithm can be made to run in linear time even more simply, by using a d-ary heap in place of Fibonacci heap.

KRUSKAL'S ALGORITHM:

Minimum Spanning Tree:

- Given a **connected** and **undirected** graph, a spanning tree of that graph is a sub-graph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A **Minimum Spanning Tree (MST)** or **minimum weighted spanning tree** for a weighted, connected and undirected graph is **a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of spanning tree is the sum of weights given to each edge of the spanning tree.**
- A minimum spanning tree has **(V-1)** edges where **V** is the number of vertices in the given graph.

Steps for finding MST using Kruskal's Algorithm:

1. Sort all the edges in ascending order of their weights
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge, else discard it. [This step uses **Union-Find Algorithm** to detect cycle.]
3. Repeat step #2 until there are (V-1) edges in the spanning tree.

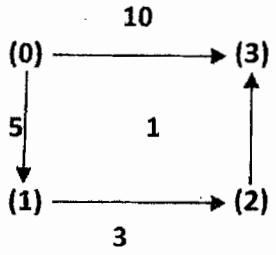
- The algorithm is **Greedy Algorithm.** The Greedy choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

- **Time Complexity: O(E log E) or O(E log V).**
- Sorting of edges takes O(E log E) time. After sorting, we iterate through all edges and apply find-union algorithm.
- The find-union operation can take almost O(log V) time.
- So overall time complexity is: O(E log E + E log V) time
- The value of E can be at most $O(V^2)$, so O(log V) and O(log E) are same, therefore overall time complexity is: O(E log V).

Floyd And Warshall's Algorithm:

- A weighted directed graph is a collection of vertices connected by weighted edges (where as weight is some real number).
- This algorithm is used for solving the All Pair Shortest Path Problem.
- The problem is to find shortest distances between every pair of vertices in a given edge weighted directed graph.
- Dijsktra's Algorithm doesn't works with negative edges.
- Note that value of **graph[i][j] = 0: if i == j**, and **graph[i][j] = ∞ (infinity)**: if there is no direct edge from vertex i to j.

Input Graph:



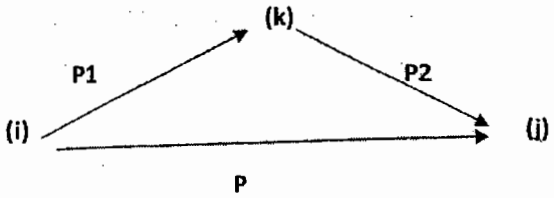
Input:
Matrix Representation:

	0	1	2	3
0	0	5	∞	10
1	∞	0	3	∞
2	∞	∞	0	1
3	∞	∞	∞	0

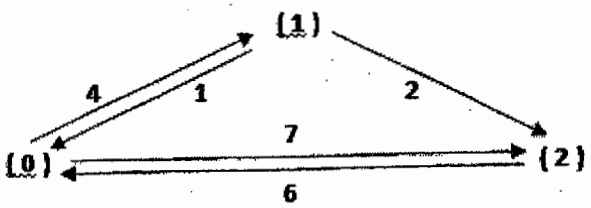
Output:
Shortest Distance Matrix:

	0	1	2	3
0	0	5	8	9
1	∞	0	3	4
2	∞	∞	0	1
3	∞	∞	∞	0

- We initialize the solution matrix same as the input graph matrix as a first step.
- Then we update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.
  1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
  2) k is an intermediate vertex in shortest path from i to j.
  We update the value of dist[i][j] = dist[i][k] + dist[k][j], if dist[i][j] >dist[i][k] + dist[k][j]



Example:



	0	1	2
0	0	4	7
1	1	0	2
2	6	∞	0

$D^{(0)}$ = 0 4 7
1 0 2 => Original Weights
6 ∞ 0

$D^{(1)}$ = 0 4 7
1 0 2 => Consider vertex 1: D(2,1)=D(2,0)+D(0,1)=10
6 10 0

$D^{(2)}$ = 0 4 6
1 0 2 => Consider vertex 2: D(0,2)=D(0,1)+D(1,2)=6
6 10 0

$D^{(3)}$ = 0 4 6
1 0 2 => Consider vertex 3: No Changes
6 10 0

- Looking at this example, we can come up with the following algorithm:
- Let D store the matrix with the initial graph edge information initially, and update D with the calculated shortest paths.

```
for( k = 1 ; k < n ; k++ ){
    for( i = 1 ; i < n ; i++ ){
        for( j = 1 ; j < n ; j++ ){
            D[i][j] = min( D[i][j], D[i][k]+D[k][j] )
        }
```

- **The final D matrix will store all the shortest paths.**

Time Complexity: $O(V^3)$

Bellman-Ford Algorithm:

- Given a graph and a source vertex src in graph, **find shortest paths from src to all vertices in the given graph. The graph may contains negative edges.**

- We have discussed Dijkstra's algorithm for this problem. **Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VlogV).**
- **Dijkstra's doesn't work for graphs with negative edges, Bellman-Ford works for such graphs.**
- Bellman-Ford is also simpler than Dijkstra and suits well for distributed systems, but Time Complexity of **Bellman-Ford is O(VE)**, which is more than Dijkstra.

Algorithm:
Following are the detailed steps:
Input: Graph and Source vertex src

Output: Shortest distance to all vertices from src. If there is negative weight cycle, then shortest distance are not calculated, negative weight cycles is reported.

Step 1:

- This step initializes distances from source to all vertices as infinite and distance to source itself as 0.
- Create an array **dist[]** of size **|V|** with all values as infinite except dist[src], where src is source vertex.

Step 2:

- This step calculates shortest distances: do following **|V|-1** times, where |V| is the number of vertices in given graph.
- do following for each edge: u-v
  if(dist[v] >dist[u] + weight(u, v))
  dist[v] = dist[u] + weight(u, v)

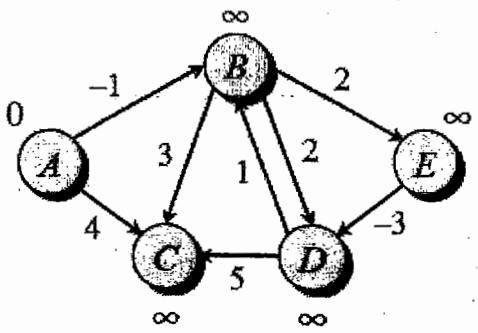
Step 3:

- This step reports if there is a negative weight cycle in graph. Do following for each edge u-v: if(dist[v] >dist[u] + weight(u, v)) -> then graph contains negative weight cycle

- the idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contains negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

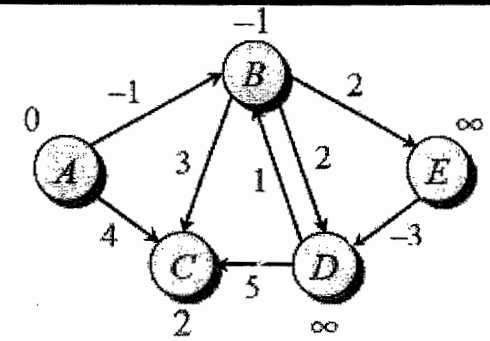
Example:

- Let the given source be 0. Initialize all distances as infinite, except the distance to source itself. Total no. of vertices in the graph is 5, so all edges must be processed 4 times.



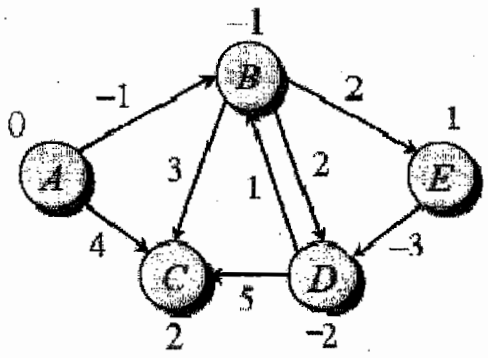
A	B	C	D	E
0	∞	∞	∞	∞

- We get following distances when all edges are processed first time. The first row in figure shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

- **The first iteration guarantees to give all shortest paths which are at most 1 edge long.** We get following distances when all edges are processed second time(the last row shows final values)



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

- **The second iteration guarantees to give all shortest paths which at most 2 edges long.** The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

A* Search Algorithm:

- A* Search is the one of the best and popular technique used in path finding and graph traversals.
- This is the most interesting part of A* search algorithm, that is it used in games.
- A* search algorithm is often used to find the shortest path from one point to another point. You can use this for each enemy to find a path to the goal.
- The worst case time complexity is O(E), where E is the number of edges in the graph.
- Auxiliary space in the worst case we can have all the edges inside the open list, so required auxiliary space in worst case is O(V), where V is the total number of vertices.

Johnson's Algorithm for All-pairs shortest paths:

- The problem is to find shortest paths between every pair of vertices in a given weighted directed graph and weights may be negative.

- We have discussed Floyd Warshall's algorithm for this problem, using this algorithm, we can find all pair shortest paths in **$O(V^2 \log V + VE)$** time.
- Johnson's algorithm uses both **Dijkstra's** and **Bellman-Ford** as subroutines.
- The idea of Johnson's algorithm is to reweight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.