Classification of Language

1. Procedure Oriented Language:

Language in which more focus is given on function (procedure) rather than data, such type of language is called procedure oriented language.

All procedure oriented language follows top down approach. Approach in which first we think about function and then data, is called procedure oriented language.

Example: FORTRAN (which is the first procedure oriented language), C, PASCAL.

2. Object Oriented Language:

Language in which more focus is given on data (data) rather than the function, such type of language is called object oriented language.

All object oriented language follows bottom down approach. Approach in which first we about data and then function, is called object oriented language.

Concept of OOP in 1960

Example: Simula, C++, smaltak, java.

Simula is the first object oriented language and smaltak is the only pure object oriented language.

C++ is not pure object oriented language. It is called partial object oriented language.

3. Object Based Programming Language:

Languages which support some features of object oriented language but not all, such language is called object oriented language.

Example: visual, Ada (first object based programming language)

4. Rules Based Programming Language:

Example: PROLOG and LISP.

Advantages of C Language

1. C is portable.
2. C is efficient. It can interact with hardware efficiently.
3. C is flexible.
4. C is freely available. Wide variety of compiler available.

Any New Language Is Basically Design for Two Reasons

1. To overcome or avoid limitation of previous language.
2. To provide new feature which are not available on existing languages

Limitation C With Respect To C++

1. Language doesn’t provide security for data.
2. Using function can achieve code reusability but reusability is limited.
3. The programs are not extendible.
4. We can’t write function inside structure.
5. When program become complex, understanding and maintaining such program is very difficult.

“Bjarne Stroustrup” design a new language C with classes in 1979 on DEC PD11 machine.

Reconstructed by ANSI on 1983

63 keywords are available on C++ language

27+5=32 ---------ANSI C

32(C) +31=63---------ANSI C++

Token in C:

Identifiers, keywords, operator, string, special symbol, constant

Difference between Procedure Oriented Language and Object Oriented Language

Procedure oriented language	Object oriented language
1. Emphasis on steps or algorithm	1. Emphasis on data of the program
2. Programs are divided into small code units i.e. function	2. Programs are divided into small data units i.e. classes
3. Most function shared global data and can be modified.	3. Data is hidden and not accessible outside classes
4. Data move from function to function	4. Objects communicate with each other
5. Top down approach	5. Bottom up approach

Variable Declaration

In C language, variable should be declare at the starting of the block.

This restriction is removed in C++. In C++, we can declare variable anywhere in function.

Data Types:

C++ support all the data types provided by C like int, float etc. C++ added two data types:

1. bool: It can take true or false value. 1 bye ASCII format is for char

2. wchar_t: It can store 16 bit character. Its take 2 bytes in memory. 2 bytes UNICODE format is for wchar_t

Structure

It is a collection of similar and dissimilar data. It is used to bind logically related data into single unit.

This data can be modified by any function to which the structure is passed. Thus there is no security provided for the data within the structure. This concept is modified in C++ to bind data and as well function.

Difference between Structure in C and C++

Structure in C	Structure in C++
1. At the time of creating of structure variable, struct keyword is compulsory. Example: struct time t	1. At the time of creating object of structure, struct keyword is not compulsory. Example: time t
2. By default all the members are accessible outside the structure. C languages doesn’t have concept of access specifier	2. By default all members of structure in C++ are public. We can make them private.

Access Specifiers

By default all members in the structure are accessible in the program using dot and arrow operator. Such access can be restricted by access specifier.

Private: Accessible only within the structure

Public: Accessible within and outside structure.

Function Overloading

Function with same name but different in signature are called function overloading. Return type is not for function over loading.

Function call is resolved according to the type of arguments passed.

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

void sum(int num1,int num2)
{
        cout<<endl<<"Sum Is::::"<<num1+num2<<endl;
}


void sum(int num1,float num2)
{
        cout<<endl<<"Sum Is::::"<<num1+num2<<endl;
}

void sum(int num1,int num2, int num3)
{
        cout<<endl<<"Sum Is:::::"<<num1+num2+num3<<endl;
}
int main(void)
{
        sum(100,200); // this will call void sum(int num1,int num2)

        sum(100,200.0f); // this will call void sum(int num1,float num2)

        sum(100,200,300); // this will call void sum(int num1,int num2, int num3)
        return 0;
}
```

Function Overloading

Function having same name but different either types of arguments, actual number of arguments, ordered of arguments, such process of writing function is called function overloading.

1. Function having same name but different in number of arguments

```
Example: void sum(int num1,int num2)

               Void sum(int num1,int num2,int num3)
```

2. Function having same name and arguments but different in types of arguments

```
Example: void sum(int num1,int num2)

            void sum(int num1,float num2)
```

3. Function having same name and number of arguments but order of arguments are different

```
Example: void sum(int num1,float num2)

           void sum(float num1,int num2)
```

Name Mangling

When we write function in C++, the compiler internally create unique name for each and every function by looking towards name of function and type of arguments pass that function. Such process of creating unique name is called name mangling. That individual name is called mangled name.

```
Example: void sum(int num1,int num2)

                  sum@ num1,num2

              void sum(int num1,int num2,int num3)

          sum@ num1, num2, num3
```

cin and cout

C++ provides an easier way for I/P and O/P.

The O/P

```
cout<<"Hello C++";
```

The I/P

```
cin>>var
```

```
#include<stdio.h>

int main(void)
{
        printf("Hello World");

        return 0;
}
```

printf is a function and declared in stdio.h header file.

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

int main(void)
{
        cout<<"Hello World";
        return 0;
}
```

cout is a object of ostream class and ostream class is declared in iostream.h. That why if you want to use cout object it is necessary to include iostream.h header file.

The operator which is used with cout is called insertion operator <<).

```
Int res=30;

cout<<”Res=”<<res;
```

```
#include<stdio.h>
int main(void)
{
        int num;
        scanf("%d",&num);
        return 0;
}
```

scanf is a function and declared in stdio.h file. If you want to use scanf, it is necessary to include stdio.h file.

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

int main(void)
{
        int num;
        cin>>num;
        return 0;
}
```

cin is a object of istream class and istream class is declared in iostream.h header file. If you want to use cin object, it is necessary to include iostream.h header file.

The operator which is used with cin is known as extraction operator.

Class

Building block that bind together data and code

Program is divided into different classes.

Class has

1. Variables(data member)
2. Functions(member function or method)

By default class member are private. Not accessible outside class.

Classes are standalone component and can be distributed inform of libraries.

Class is a blue print of an object.

Object

Object is an instance of a class.

Entity that has physical existence, can store data, send and receive message to communicate with other object.

Object has

1. Data member(state of object)
2. Member function(behavior of object)
3. Unique address

Object has three characteristics

1. State
2. Behavior
3. Identity

State

The value stored in the data member of the object is called state of object.

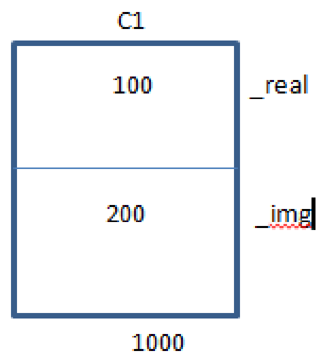
Behavior

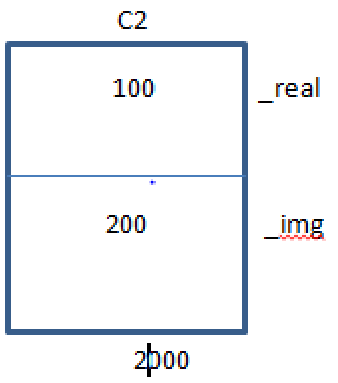
Behavior is how object acts and reacts when its state is changed and operations are done. Behavior is decided by the member function. Operation performed is also known as messages.

Note: when you create object of empty class, it will take 1 byte.

Identity

Every object has characteristics that makes object unique. Every object has unique address. Identity of C1 is 1000 and c2 is 2000





Data Member

Data member of the class are generally made as private to provide data security. The private member can’t be accessed outside the class. So the data members are always accessed by the member function.

Constructor

Initialize the object.

Destructor

De-initialize the object.

Mutators
Modifies the state of object

Example: setMethod, setSal

Inspector
Don’t modify state of object

Example: getMethod, getSal

Facilator
Facilator provide functionality like IO.

Constructor
It is member function of a class having same name as that of a class and doesn’t have any return type not even void

Constructor gets automatically call when object is created that is memory is allocated to the object.

If we don’t write constructor, compiler provides a default constructor.

We can have constructor with

1. No arguments: Initialize data member with default value.
2. One or more arguments: Initialize data member to values passed to it.

Constructor Overloading
Constructor having different signature, such constructor overloading is called as constructor overloading.

We can overload constructor. Return type is not considered at the time of function overloading.

Why Constructor Doesn’t Have Any Return Values

Since the job of constructor is to initialize data member of class, there is no need to return any values from constructor. That’s why constructor doesn’t have any return values.

Three Types of Constructor

1. Default constructor.
2. Parameter less or No argument constructor
3. Parameterized / argument constructor

Note: copy constructor comes under parameterized constructor.

Destructor

It is a member function of a class having same name as that of class proceeded with ~ sign and don’t have any return type and arguments.
Destructor gets automatically called when object is going to destroy i.e. memory is going to destroy.

If we don’t write destructor compiler will provide default destructor

Generally we are implementing destructor when constructor of the object is allocating any resource.

Example: We have a pointer as data member which is pointing dynamically allocated memory.

Virtual Destructor

If your base class destructor is virtual then objects will be destructed in a order(firstly derived object then base). If your base class destructor is not virtual then only base class object will get deleted (because pointer is of base class "Base *myObj"). So there will be memory leak for derived object.

The construction of derived object follows the construction rule but when we delete the "b" pointer (base pointer) we have found that only the base destructor is call. But this must not be happened. To do the appropriate thing we have to make the base destructor virtual. Now let see what happen in the following

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

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

```
class base
{

public:
   base(){cout<<"Base Constructor Called\n";}
   virtual ~base(){cout<<"Base Destructor called\n";}

};
class derived1:public base
{

public:
   derived1(){cout<<"Derived constructor called\n";}
   ~derived1(){cout<<"Derived destructor called\n";}

};
int main()
{

   base* b;
   b=new derived1;
   delete b;

}
```

So the destruction of base pointer (which take an allocation on derived object!) follow the destruction rule i.e. first the derived then the base. On the other hand for constructor there are nothing like virtual constructor.

Pure Virtual Destructor

A destructor can be declared virtual or pure virtual; if any objects of that class or any derived class are created in the program, the destructor shall be define.

A class with a pure virtual destructor is an abstract class.

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

#include <iostream>
using namespace std;
class base
{

public:
   base(){cout<<"Base Constructor Called\n";}
   virtual ~base()=0;

};
base::~base()
{
        cout<<endl<<"Inside Pure Virtual Destructor"<<endl;
}
```

```
class derived1:public base
{

public:
   derived1(){cout<<"Derived constructor called\n";}
   ~derived1(){cout<<"Derived destructor called\n";}

};
int main()
{

   base* b;
   b=new derived1;
   delete b;

}
```

This Pointer

When we call member function by using object, implicitly one argument is passed to that function; such argument is called this pointer.

This is a keyword in C++.

This pointer always store address of current object or calling object.

Thus every member function of the class receives this pointer which is the first argument of that function.

This pointer is constant pointer.

```
Example: class-name * const this;

           Tcomplex * const this;
```

Can We Do Destructor Overloading

No, since the job of destructor is to de-initialize data member of the class, there is no need to pass arguments to destructor. For overloading function argument must be different, type of arguments must be different or order of argument must be different but destructor doesn’t take any arguments. That why can’t do destructor overloading

The Size of Empty Object Is 1 Byte

When you create object of a class its gets three characteristics:

1. State
2. Behavior

3. Identity

When you create object of empty class, at that time state of object is nothing. Behavior of object is also nothing. But that object has unique entity that is its address. Memory in computer is always organized in the form of bytes. Byte is a unit of memory. Minimum memory at object unique address is 1 byte. That’s why size of empty object is 1 byte.

Default Arguments

In C++, function may have arguments with the default value. By this argument while calling a function is optional. If such argument is not passed, then its default value is considered otherwise argument is treated as normal argument. Default argument should be given in right to left order.

```
Example:  int sum(int num1=100,int num2=200,int num3=300)

          int sum(int num1,int num2=200,int num3=300)
```

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


void sum(int num1=100,int num2=200,int num3=300)
{
        cout<<endl<<"Result Is:::"<<num1+num2+num3<<endl;
}

int main(void)
{
        sum(1000,2000,3000);
        sum(100,200);
        return 0;
}
```

Enum

It is a integer constant.

```
Example: enum Ecolor{Red=0,Blue,White};
```

C++ Is Not Pure Object Oriented Language

C++ allows writing global function and main itself is a global function.

We can access the private data member of the class outside the class using pointer (reinterpret cast).

In pure object oriented language for built in data types predefined classes are available. Such classes are called wrapper classes. Concepts of wrapper classes are not available in C++.

We can access the private data member of the class outside the class using friend function or friend class.

Difference between Function and Macro

Function	Macro
1. Functions are called at runtime by creating function activation record.	1. Macro are replaced before compilation
2. Due to runtime call functions are slow	2. Macros are much faster as they replace before compilation
3. Function can be recursive	3. Macros can’t be recursive
4. Calling same function multiple times will not increase the size of file.	4. Calling macros multiple times will replace it multiple times and will cause increase in size of file

Inline Function

C++ provides a keyword that makes function as inline function. Inline function gets replace by the compiler at its call statement. It ensures faster execution of the function just like macro.

Advantage of inline function over macro

Inline functions are type safe.

Inline is a request made to compiler while macro is a command. Every function may not be replaced by compiler rather it avoids replacement if it contains cases like function contacting switch loop or recursion may not be replaced.

So generally mutator and inspector are made as inline function (seta and geta method).

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

```
class Treal
{
public:
	int _real,_img;

	Treal()
	{
		this->_img=0;
		this->_real=0;
	}
	Treal(int real,int img)
	{
		this->_real=real;
		this->_img=img;
	}

	void DisplayOutput(void)
	{
		cout<<endl<<"Real No Is:::"<<this->_real<<"\t"<<"Img No Is:::"<<this-
>_img<<endl;
	}

	inline void SetRealNumber(int number)// Mutator
	{
		this->_real=number;
	}

	inline int GetRealNumber() // Inspector
	{
		return this->_real;
	}
};

int main(void)
{
	Treal obj(100,200);
	obj.DisplayOutput();

	obj.SetRealNumber(999);
	obj.GetRealNumber();
	obj.DisplayOutput();
	return 0;
}
```

Anonymous Object:

Object with no name is called as anonymous object.

Example: Tcomplex(100,200) ;

Tcomplex C
In this case parameter less constructor will be called.

Tcomplex C(100,200)
In this case parameterized constructor which takes 2 arguments will be called.

Tcomplex C(300)
In this case parameterized constructor which takes 1 argument will be called.

Tcomplex C=400
When we write a statement like this compiler internally converts Tcomplex C(400). In this case, parameterize constructor which takes 1 argument will be called.

Tcomplex(400,600)
In this case, anonymous object is created and parameterize constructor having two arguments will be called.

Tcomplex C()
In this case, compilers consider C as function name and Tcomplex as a return type. So in this case any kind of constructor will not be called and this is not any kind of compile time error.

Tcomplex C=(600,700)
Comma operators always consider last value. So 700 will be assigned to C and implicitly compiler will give call to single argument constructor.

Tcomplex C{700,800}
Compile time error

1. C is local function identifier are illegal
2. C is not a function

Dynamic Memory Allocation
In C programming, we can use malloc ,realloc function to allocate memory dynamically. These functions are declared in stdlib.h header. When we are

allocating memory dynamically, we should free that memory after use, using free function.

Malloc return a void pointer. Void pointer is a pointer whose scale factor we don't know. If we don't know the scale factor, we can't do pointer arithmetic on it. So we have to type casted depending upon that data type for which we are allocating memory.

Allocating Memory for Variable Statically

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


int main(void)
{
int num;
        int num=200;
        return 0;
}
```

num will get memory on the stack segment.

Allocating Memory Dynamically for a Single Variable Using Malloc Function

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


int main(void)
{
        int *num=NULL;
        num=(int*)malloc(sizeof(int));
        *num=200;
        return 0;
}
```

Allocation of Memory for Array Statically

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


int main(void)
{
        int arr[5];
        return 0;
}
```

Allocating Memory Dynamically for an Array Using Malloc Function

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

```
int main(void)
{
        int *num=NULL;
        int size=5;
        num=(int*)malloc(sizeof(int)*size);

        return 0;
}
```

C++ provides operator new and delete for allocating and de-allocating memory at runtime. The memory is allocated from heap.

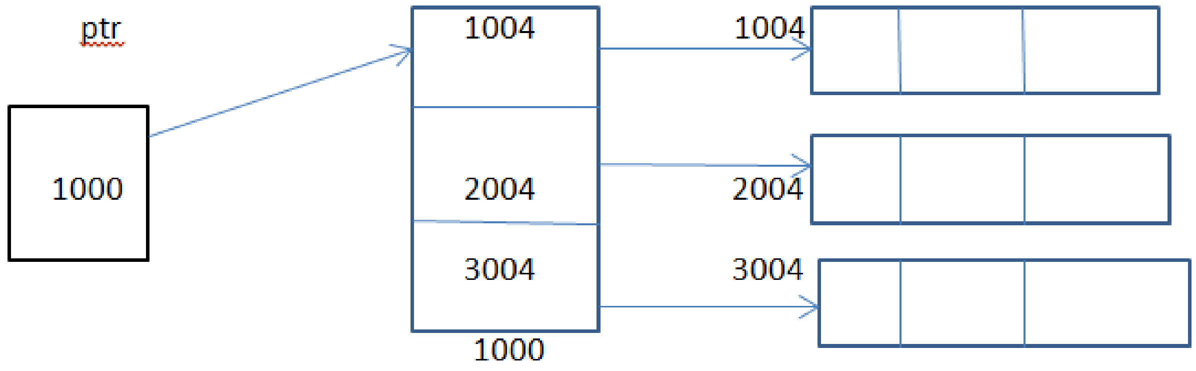
Allocating Memory for a Single Variable Using new Operator

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


int main(void)
{
        int *num=NULL;
        num=new int;
        *num=200;
        return 0;
}
```

Allocating Memory for an Array Using new Operator

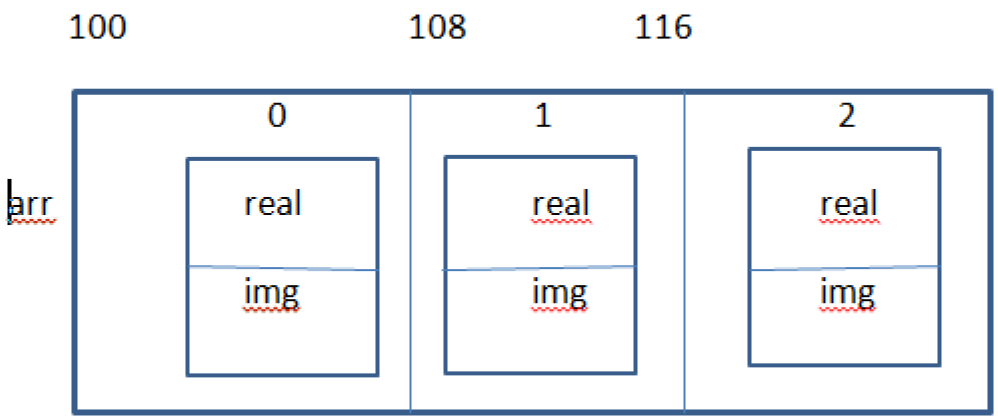
```
#include<iostream>
using namespace std;


int main(void)
{
        int *num=NULL;
        int size=5;
        num=new int[size];
        delete []num;
        return 0;
}
```

2D Dynamic Memory Allocation Diagram



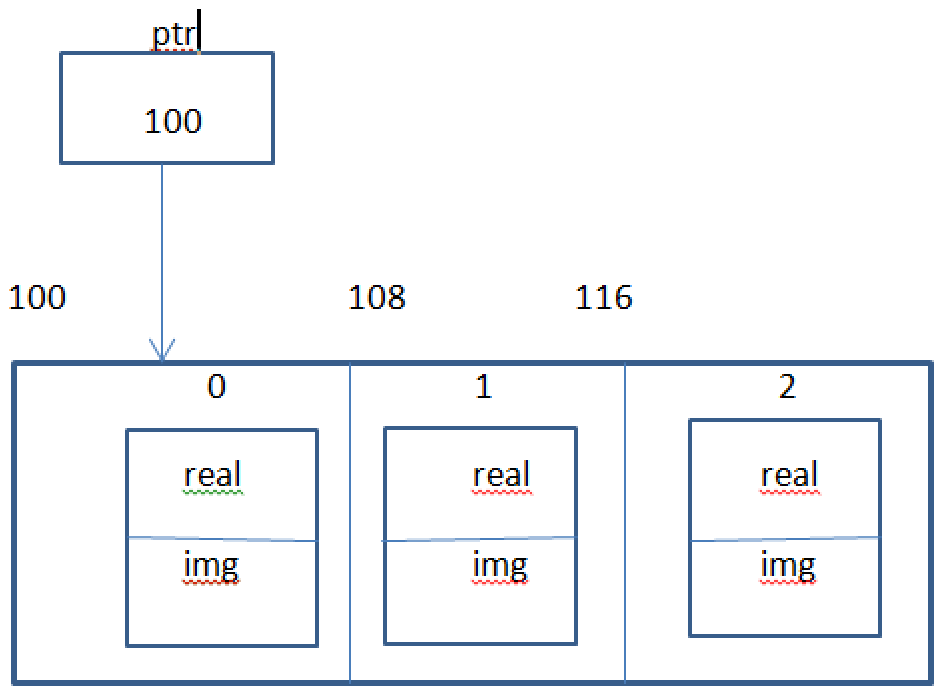
Static Array of Object

```
Tcomplex arr[3];

arr[0].DisplayOutput();
```



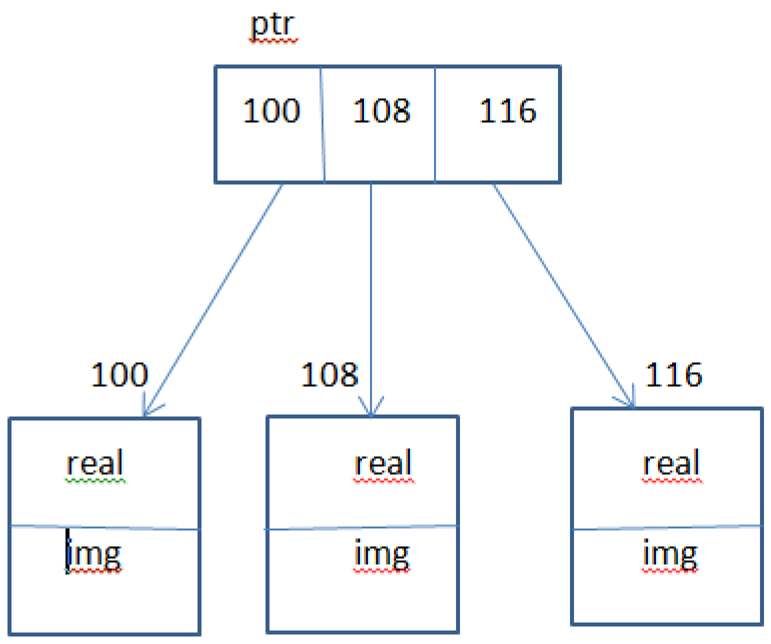
Dynamic Array of Object

```
Tcomplex *ptr=new Tcomplex[3];

ptr[0]->DisplayOutput();
```



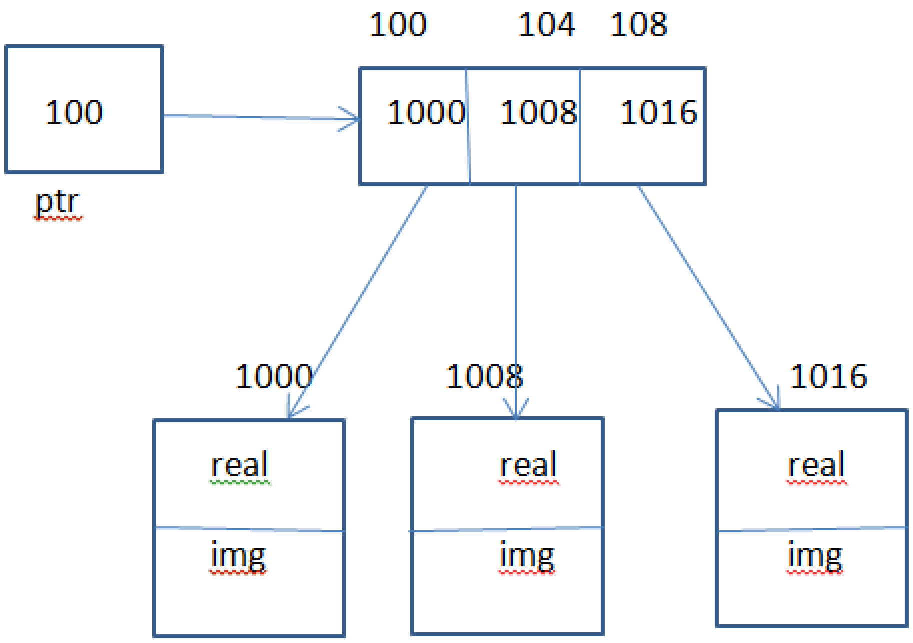
Array of Pointer

```
Tcomplex *ptr[3];

for(int i=0;i<3;i++)

{

        ptr[i]=new Tcomplex;

}
```



Pointer to Pointer Object

```
Tcomplex **ptr=NULL;

ptr=new Tcomplex*[3];

for(int i=0;i<3;i++)

{

        ptr[i]=new Tcomplex;

}
```



Difference between malloc and new

Malloc	new
1. Malloc is a function	1. New is a operator
2. Allocating memory using malloc constructor is not call i.e. malloc is not aware of constructor.	2. Allocating memory using new constructor is call i.e. new is aware of constructor
3. If malloc fails, it return NULL	
4. Need to specify number of bytes and type casting is required	3. If new fails, it throws bad_alloac exception
5. sizeof operator is required	4. Need to specify number of object and type casting is not required.
6. When we de-allocate memory by using free function, which is allocated for an object by using malloc function, at that time explicitly destructor will not call.	5. sizeof operator is not required
	6. Implicitly destructor will call.

```
Tcomplex *ptr=(Tcomplex *)malloc(sizeof(Tcomplex)*1); // are not aware of constructor
ptr->DisplayOutput();
```

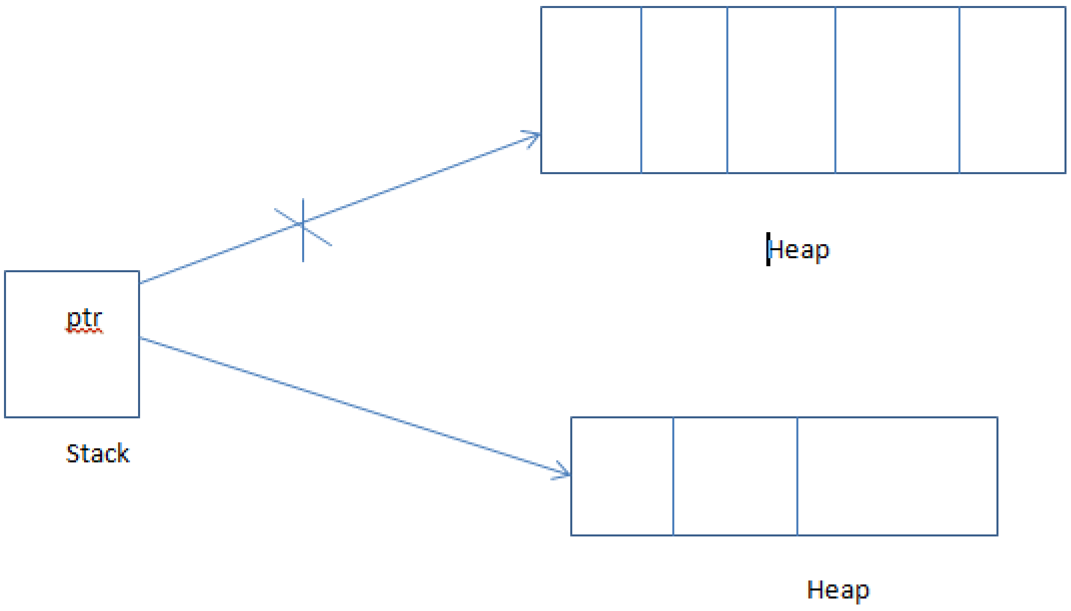
```
free(ptr);

Tcomplex *ptr=new Tcomplex; // aware of constructor
ptr->DisplayOutput();
delete ptr;
```

Note: The error which we can handled at runtime is known as exception error.

Memory Leakage

Memory allocated but not freed after used, then there is no way to reach that memory, such type of wastage of memory is called memory leakage.

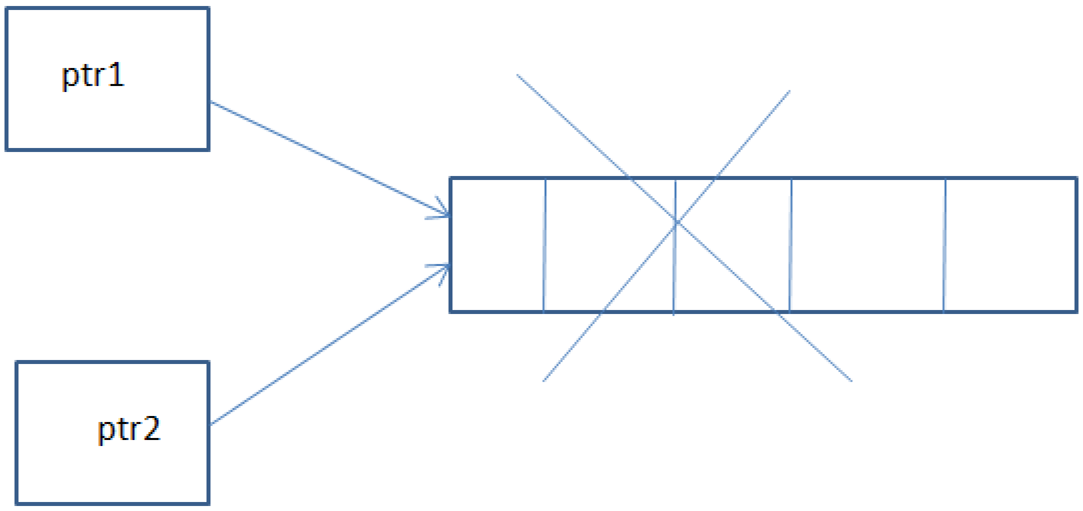
```
int *ptr=new int[5];  // This memory is Leakage
ptr=new int[3];
```



Dangling Pointer

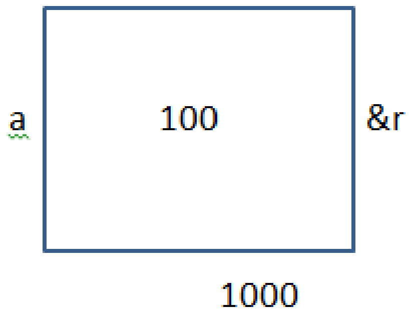
Pointer pointing to the memory which is not available, such type of pointer is called dangling pointer.

```
int *ptr1=new int[5];  // ptr becomes dangling pointer
int *ptr2=ptr;
delete ptr2;
```



References

References are treated as alias to the variable. It can be used as another name for the same variable.



```
Example:   int a=100;

            Int &r=a;
```

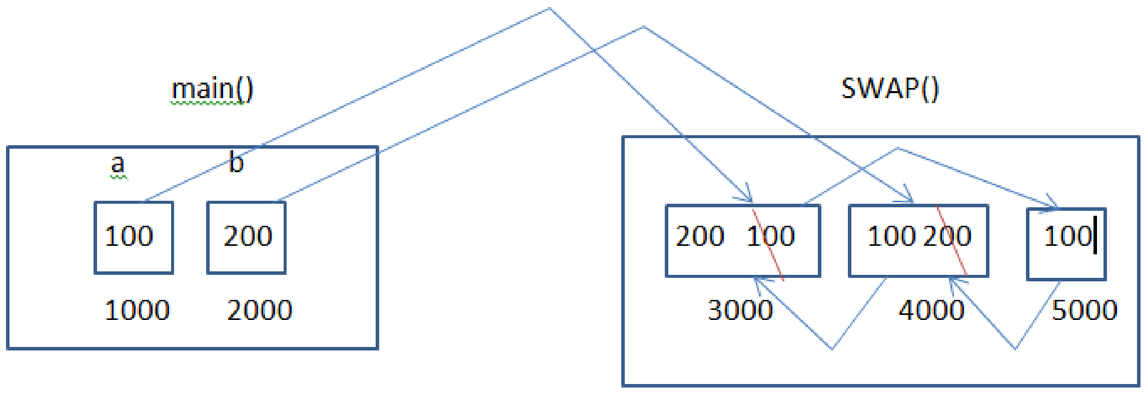
```
#include<iostream>
using namespace std;

int main(void)
{

int a=100;
int &r=a;

cout<<endl<<"a is:::"<<a<<"&r is:::"<<r<<endl;
cout<<endl<<"address of a is:::"<<&a<<"\t"<<"address of &r is:::"<<&r<<endl;

a=200;
cout<<endl<<"a is:::"<<a<<"&r is:::"<<r<<endl;
cout<<endl<<"address of a is:::"<<&a<<"\t"<<"address of &r is:::"<<&r<<endl;

        return 0;
}
```

Thus we can pass the argument to the function by values, by address or by reference.

Reference is internally treated as constant pointer to the variable which gets automatically de-referenced.

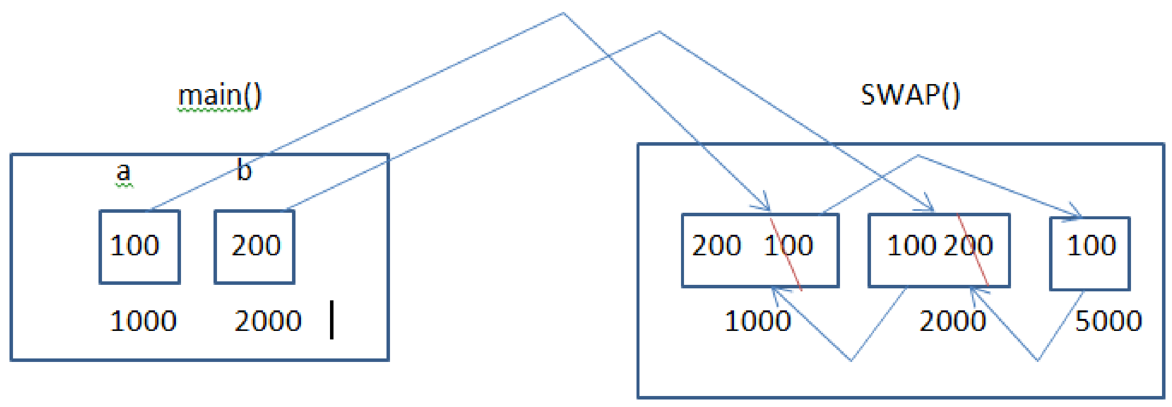
Swapping of Two Variable Using Call by Value Function

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


void SWAP(int num1,int num2)
{
        int temp;
        temp=num1;
        num1=num2;
        num2=temp;
}
int main(void)
{
        int a=200,b=100;

        cout<<endl<<"=======Before Swapping======"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        SWAP(a,b);

        cout<<endl<<"=======After Swapping======"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        return 0;
}
```



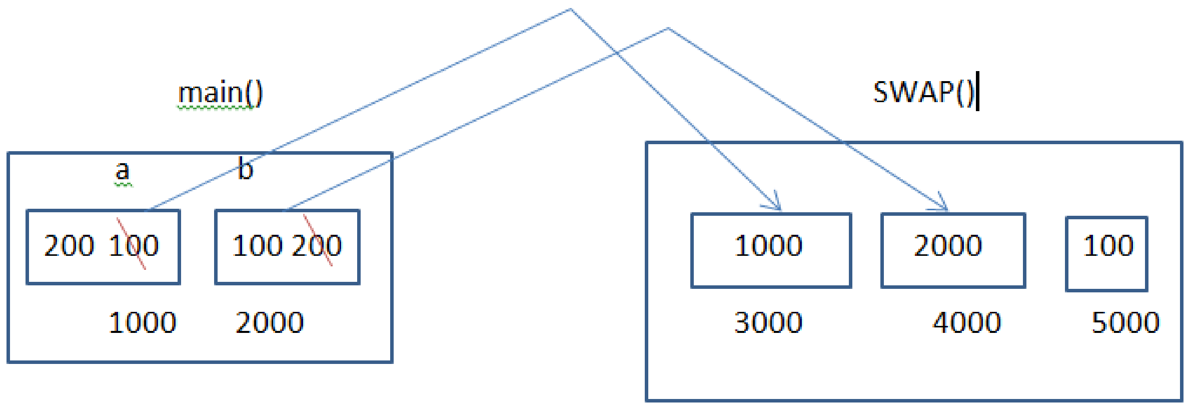
Swapping of Two Variable Using Call by Reference Function

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


void SWAP(int &num1,int &num2)
{
        int temp;
        temp=num1;
        num1=num2;
        num2=temp;
}
int main(void)
{
        int a=200,b=100;

        cout<<endl<<"=======Before Swapping======"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        SWAP(a,b);

        cout<<endl<<"=======After Swapping======"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        return 0;
}
```



Swapping of Two Variable Using Call by Address Function

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


void SWAP(int *num1,int *num2)
{
        int temp;
        temp=*num1;
        *num1=*num2;
        *num2=temp;
}
int main(void)
{
        int a=200,b=100;

        cout<<endl<<"=======Before Swapping======"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        SWAP(&a,&b);

        cout<<endl<<"=======After Swapping======"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        return 0;
}
```



Difference between Pointer and Reference

Pointer	Reference
1. Pointer may not be initialize at the point of declaration	1. Must be initialized
2. Pointer must be de-referenced	2. Reference are automatically de-referenced
3. Address store in pointer can be modified	3. Reference keeps referencing to the same variable till it goes out of scope
4. We can have pointer arithmetic operation, dangling pointer ,null pointer	4. Such concept does not exist for reference
5. We can initialize pointer to NULL	5. We can't initialized reference to NULL
6. We can create array of pointer	6. We can't create array of reference
7. We can create pointer to pointer	7. We can't create reference to reference

Swapping of Two Number

Using temp variable	Using + operator	Using * operator
Temp=Num1	Num1=Num1+Num2	Num1=Num1*Num2

Num1=Num1

Num2=Temp

Num2=Num1-Num2

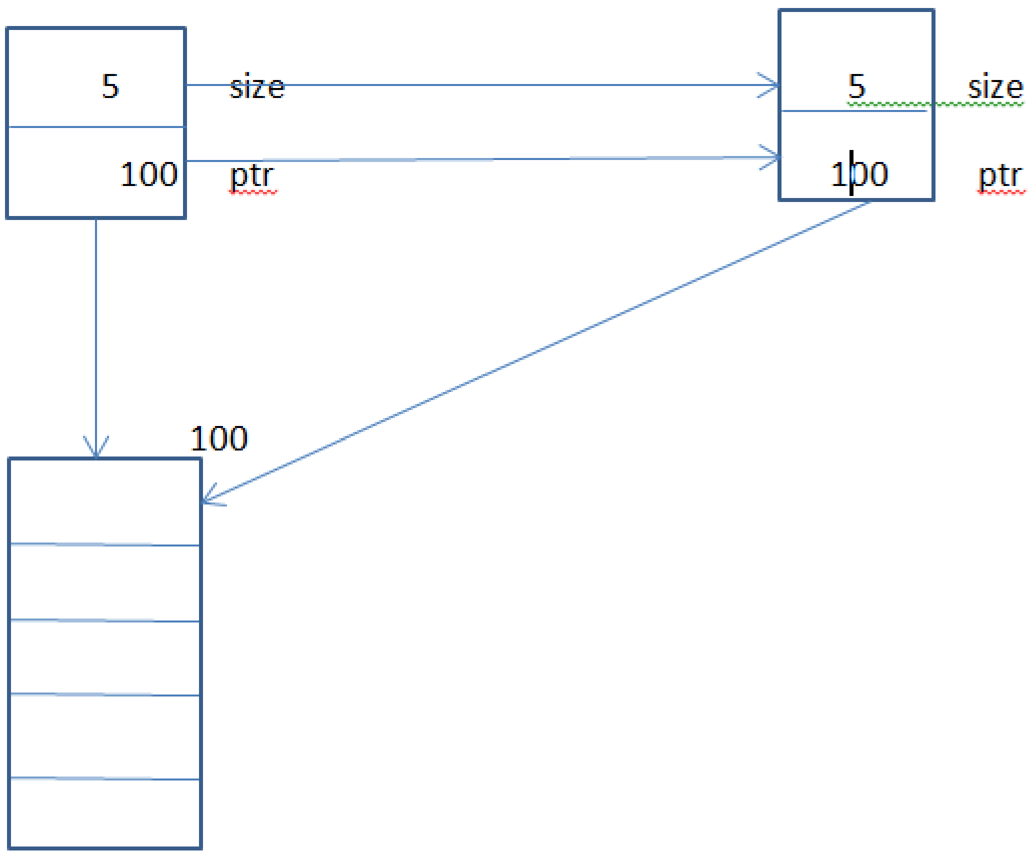
Num1=Num1-Num2

Num2=Num1/Num2

Num1=Num1/Num2

Shallow Copy

When we assign one object to another object, at that time copying all contents from source to destination object as it is. Such type of copy is called as shallow copy. By default compiler will create shallow copy. Default copy constructor always creates a shallow copy.

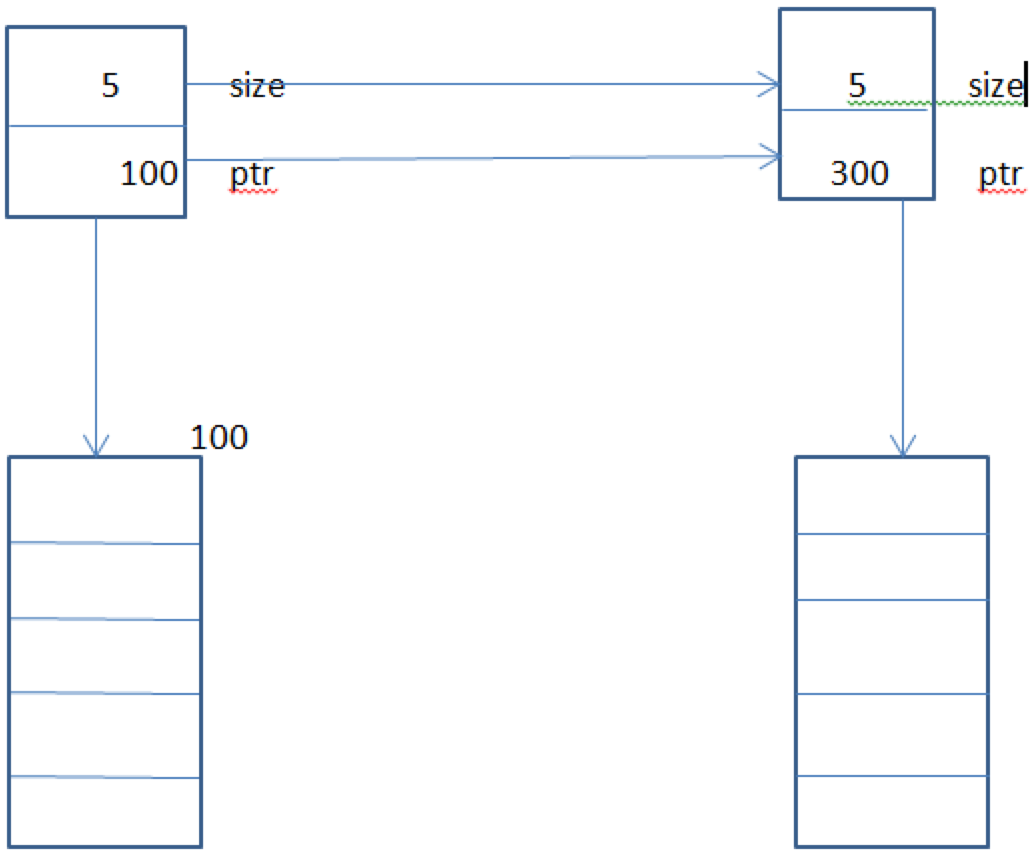


```
#include<iostream>
using namespace std;

class Tarray
```

```
{
public:
        int size;
        int *ptr;
        Tarray();
        Tarray(int size);
};

Tarray::Tarray()
{
        this->size=0;
        this->ptr=NULL;
}

Tarray::Tarray(int szie)
{
        this->size=size;
        this->ptr=new int[this->size];
}

int main(void)
{
        Tarray a(5);
        Tarray b=a; // Shallow copy will be created since class doesn't have copy constructor

        return 0;
}
```

Deep Copy

When class contains at least one data member of a pointer type, when class contain user defined destructor and when we assign one object to another object at that time, instead of coping the base address, allocate new memory for each and every object. Then copy the content from the source object into memory of destination object. Such type of copy is called deep copy.



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

class Tarray
{
public:
	int size;
	int *ptr;
	Tarray();
	Tarray(int size);
	Tarray(const Tarray &other);
};

Tarray::Tarray()
{
	this->size=0;
	this->ptr=NULL;
}

Tarray::Tarray(int szie)
{
	this->size=size;
	this->ptr=new int[this->size];
}
```

```
Tarray::Tarray(const Tarray &other)
{
	this->size=other.size;
	this->ptr=new int[this->size];
	for(int i=0;i< this->size;i++)
	{
		this->ptr[i]=other.ptr[i];
	}
}

int main(void)
{
	Tarray a(5);
	Tarray b=a; // Deep copy will be created since class contain copy constructor

	return 0;
}
```

Condition for Deep Copy

1. Class must contain at least one data member of a pointer type.
2. Class must contain user defined destructor
3. One object must be assign into another object

Creating a deep copy is a job of programmer

```
Example:  Tarray A1(3);

          Tarray A2(A1);  // Tarray A2=A1;
```

Copy Constructor

It is a special member function of a class which is having same name as that of a class and which gets call implicitly in three conditions

1. When we pass an object to function by value
2. When we return an object from function by value
3. When we assign already created object to newly created object

Such special member function of a class is called copy constructor of the class.

Job of copy constructor is to create new object from existing object.

When class doesn’t contain copy constructor, compiler provides one copy constructor by default for that class. Such copy constructor is called default copy constructor.

It is a special member function of a class having same name of a class and which takes only one argument of same type as a reference.

Example: Tarray(const Tarray & other)

When we assign already created object to newly created object, at that time it is necessary to create deep copy in copy constructor.

When we assign already created object to already created object, at that time it is necessary to create deep copy inside overloaded assignment operator function.

Reference which is used at a time of definition of a copy constructor is always for already created object.

Steps for creating deep copy

1. Copy the size from source object to destination object.
2. By using that size, allocate new memory for destination object.
3. Then copy the content from the memory of source object into memory of destination object.

When we pass an object to the function by address or by reference at that time copy constructor will not be called.

```
Tarray::Tarray(const Tarray &other)
{
	this->size=other.size;
	this->ptr=new int[this->size];
	for(int i=0;i< this->size;i++)
	{
		this->ptr[i]=other.ptr[i];
	}
}
```

Copy constructor automatically gets call when

1. Initializing one object to another object
2. Object is passed by value to function
3. Object is returned by value from function

If you want to write the definition of a member function outside class, you have to use scope resolution operator by following syntax.

Syntax: return type class-name:: function-name (parameter)

```
Tarray::Tarray(int szie)
{
	this->size=size;
	this->ptr=new int[this->size];
}
```

Can a object be passed as value to the copy constructor

It is necessary to pass object as reference and not by value because if you pass it by value its copy is constructed using the copy constructor. This means the copy constructor would call itself to make copy. This process will go on until the compiler runs out of memory

Operator Overloading

Giving extension to the meaning of operator is called operator overloading.

By using operator overloading, we can change the meaning of operator but we should not change the meaning of operator.

```
Treal Treal::operator +(const Treal obj)
{
	Treal temp;
	temp._real=this->_real+obj._real;
	temp._img=this->_img+obj._img;
	cout<<endl<<"Inside operator +(const Treal obj)"<<endl;
	return temp;
}
```

When we write a state

C3=C1+C2;

At that time compiler will resolve call for this statement like

C3=C1.operator+(C2);

By looking towards the function, we can assume that operator+ is a member function because it is called with the object C1. This function is taking only one argument C2. It means that we should pass parameter to operator+ function. The result is store in C3 object. That’s why operator+ function must return value.

C3=C1-C2; // C3=C1.operator-(C2);

C3=C1*C2; // C3=C1.operator*(C2);

C4=C1+C2+C3; //C4=C1.operator+(C2).operator+(C3);

Overloading operator+ Function by Using Friend Function

```
friend Treal operator+(const Treal obj1, const Treal obj2);
```

```
Treal operator +(const Treal obj1, const Treal obj2)
{
        Treal temp;
        temp._real=obj1._real+obj2._real;
        temp._img=obj1._img+obj2._img;
        cout<<endl<<"Inside fiend operator +(const Treal obj)"<<endl;
        return temp;
}
```

When we overload operator+ function by using friend function, at that time function call will look like

C3=C1+C2; // C3=operator+(C1,C2)

C3=C1-C2; // operator-(C1,C2)

C4=C1+C2+C3; //C4=operator+(operator+(C1,C2),C3);

What is the difference between Copy Constructor and Assignment Operator Function

When we assign already created object to newly created object, implicitly copy constructor will call.

```
Example: Tarray A1(5);

          Tarray A2=A1;
```

In this case copy constructor of A2 gets called.

When we assign already created object to already created object, implicitly assignment operator overloading function will get called.

```
Example:  Tarray A1(5), A2;

           A2=A1;
```

In this case assignment operator function will call.

When class doesn’t have assignment operator function, at that time compiler provide an assignment operator function by default. Such assignment operator is called as default assignment operator function.

When we assign already created object to newly created object at that time, we should create a deep copy in copy constructor.

When we assign already created object to already created object, at that time we should create a deep copy in assignment operator function.

Tarray A1,A2;

When we write A2=A1, implicitly call will be resolved like

A2.operator=(A1);

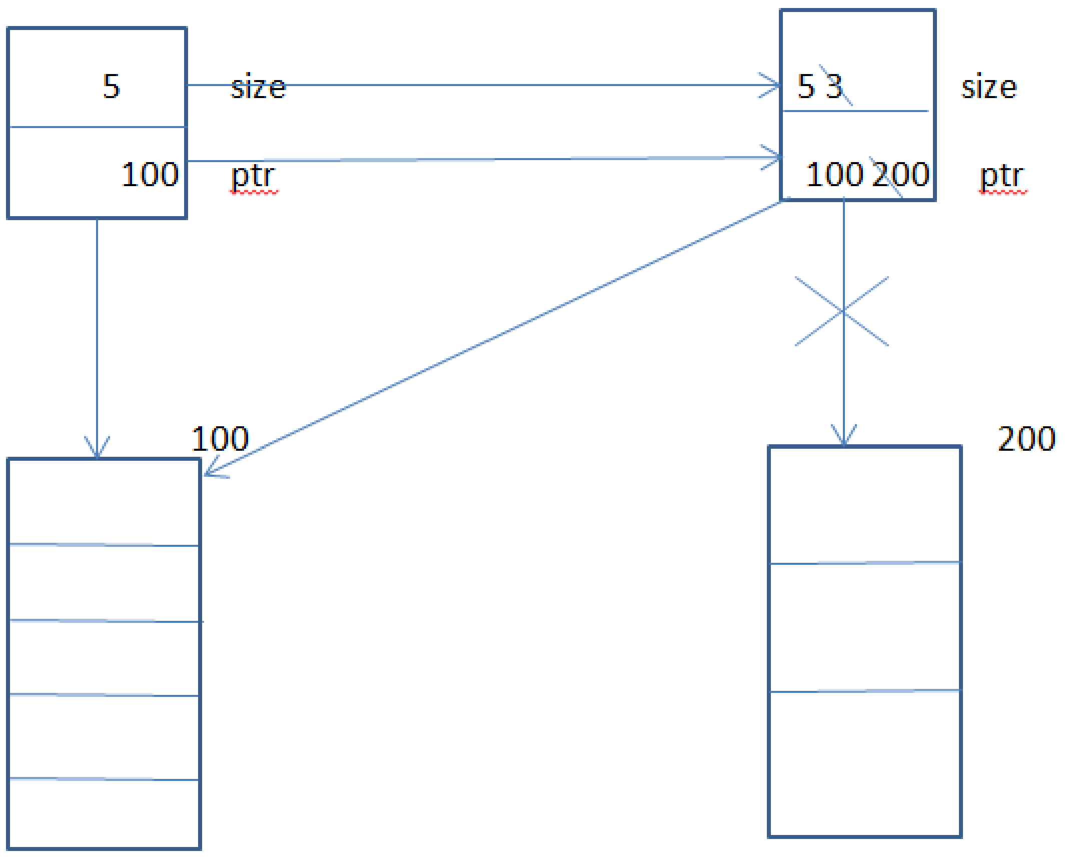
Assignment Operator Overloading Function

```
Tarray Tarray::operator=(const Tarray other)
{
	if(this==&other)
	{
		cout<<endl<<"Object is already assign"<<endl;
	}
	else if(this->size==other.size)
	{
		for(int i=0;i<this->size;i++)
		{
			this->ptr[i]=other.ptr[i];
		}
	}
	else
	{
		delete []this->ptr;

		this->size=other.size;
		this->ptr=new int[this->size];
		for(int i=0;i<this->size;i++)
		{
			this->ptr[i]=other.ptr[i];
		}
	}
	return *this;

}
```

Shallow Copy in Case of Assignment Operator Function

Tarray A1(5),A2(3);

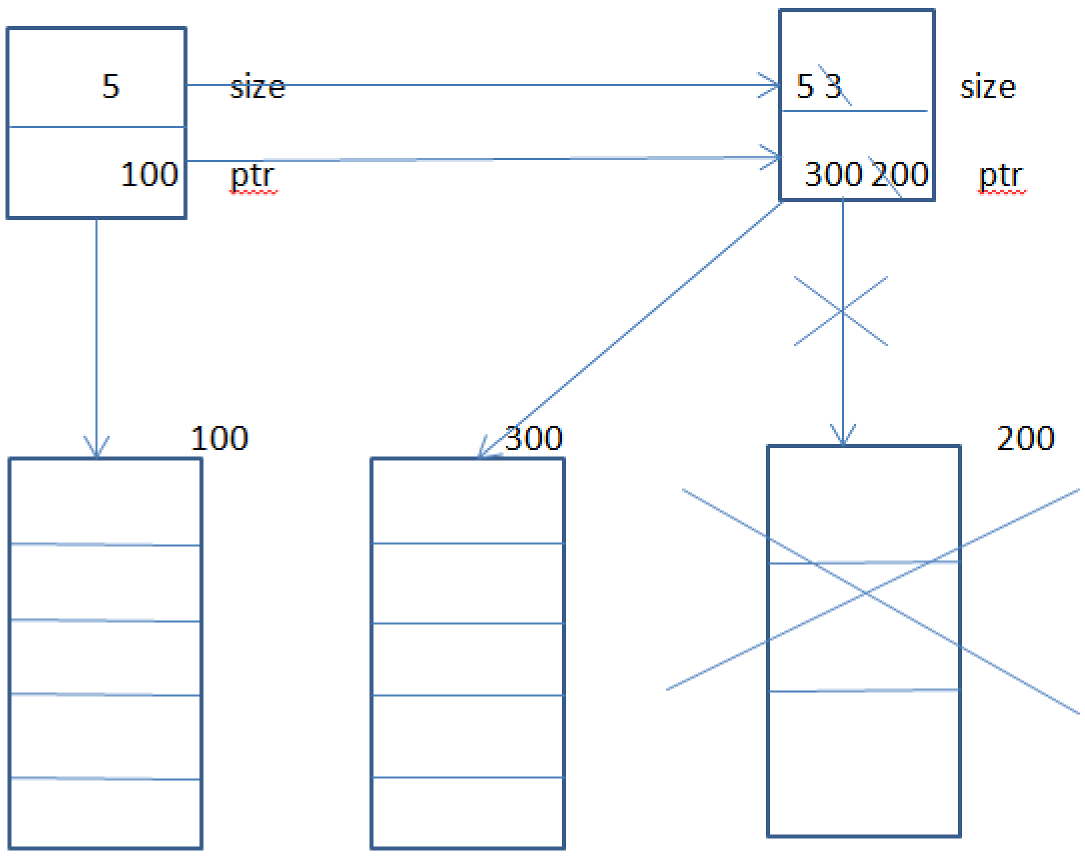
A2=A1;// A2.operator=(A2);



Deep Copy in Case of Assignment Operator Function

Tarray A1(5),A2(3);

A2=A1; // A2.operator=(A2);



List of Function that the Compiler provides by default to any Class if it is not available

1. Default parameter less constructor
2. Default parameterized constructor
3. Default copy constructor
4. Default assignment operator function
5. Default destructor

Limitation of Operator Overloading

In C++, there are some operators that we can’t overload using member function as well as friend function

1. . (dot) member selection operator

2. .* pointer to member selection
3. :: scope resolution operator
4. Sizeof size of operator
5. ?: conditional operator
6. Type id operator
7. Static cast operator
8. Dynamic cast operator
9. Const cast operator
10. Reinterpret cast operatr

There are some functions which we can't overload them as a friend function

1. = assignment operator
2. [] subscript or index operator
3. () function call
4. -> arrow operator

Friend Function

If you want to access the private data member of class inside nonmember function of that class, at that time we can declared that nonmember function as a friend.

It is a nonmember function or global function which access private member of a class in which we declared nonmember function as a friend.

Friend doesn't have this pointer

We declare global function as a friend

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

class Treal
{
private:
	int _real,_img;
public:
	Treal()
	{
		this->_real=0;
```

```
			this->_img=0;
		}

		Treal(int real,int img)
		{
			this->_real=real;
			this->_img=img;
		}

		void DisplayOutput(void)
		{
			cout<<endl<<"Real No Is:::"<<this->_real<<"\t"<<"Img No Is:::"<<this->_img<<endl;
		}
		~Treal()
		{

		}
		friend void sum();  // Friend Function

};

void sum()
{
		Treal obj(100,200);
		cout<<endl<<"Sum Of Real and Img Is::::"<<obj._real+obj._img<<endl;
		obj.DisplayOutput();
}

int main(void)
{
		sum();
		return 0;
}
```

If you want to access the private member of one class inside some other class at that time declare function as a friend.

If you want to access private member of one class inside most of another class at that time of declaring function as friend, declare class a friend.

We can declare main function as a friend but we should not declare main function as friend function.

Why Friend Function Doesn’t Have This Pointer

Since friend function is not a member function of a class in which it is declared as friend and we can’t call it by using object of class in which it is declared as a friend. That why friend function doesn’t have this pointer.

Friend Class

If you want to declared all the function B class as a friend of class A, we can declared friend class B in class A.

Defining Class Member as a Friend into another Class

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

class Tsum
{
public:
	void sum();
};

class Treal
{
private:
	int _real,_img;
public:
	Treal()
	{
		this->_real=0;
		this->_img=0;
	}

	Treal(int real,int img)
	{
		this->_real=real;
		this->_img=img;
	}

	void DisplayOutput(void)
	{
		cout<<endl<<"Real No Is:::"<<this->_real<<"\t"<<"Img No Is:::"<<this-
>_img<<endl;
	}
	~Treal()
	{

	}
	friend void Tsum::sum();  // Friend Function

};


void Tsum::sum()
{
	Treal obj(100,200);
	cout<<endl<<"Sum Of Real and Img Is::::"<<obj._real+obj._img<<endl;
	obj.DisplayOutput();
}
```

```
int main(void)
{
	Tsum obj;
	obj.sum();
	return 0;
}
```

Defining Class as Friend into another Class

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

class Tsum
{
public:
	void sum();
};

class Treal
{
private:
	int _real,_img;
public:
	Treal()
	{
		this->_real=0;
		this->_img=0;
	}

	Treal(int real,int img)
	{
		this->_real=real;
		this->_img=img;
	}

	void DisplayOutput(void)
	{
		cout<<endl<<"Real No Is:::"<<this->_real<<"\t"<<"Img No Is:::"<<this-
>_img<<endl;
	}
	~Treal()
	{

	}
	friend class Tsum;  // Friend class

};


void Tsum::sum()
{
	Treal obj(100,200);
	cout<<endl<<"Sum Of Real and Img Is::::"<<obj._real+obj._img<<endl;
	obj.DisplayOutput();
}

int main(void)
{
        Tsum obj;
        obj.sum();
        return 0;
}
```

Static Variable

Static variable are same as global variable but with limited scope.

int x=100;

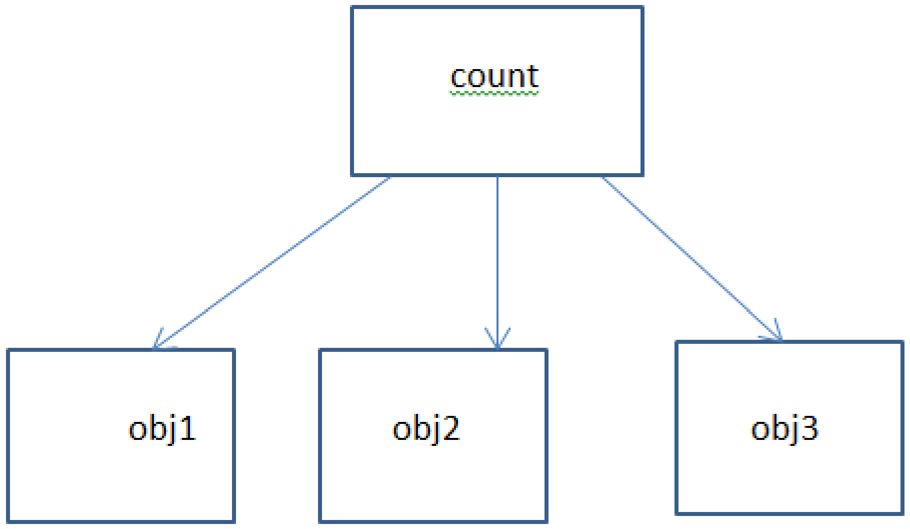
static int y=200;

x can be accessible in any file but y will be accessible only his scope.

x is nothing but share.

When we declare data member as static, it is compulsory to provide global definition for that static data member because static data member always gets memory before creation of object.

```
#include<iostream>
using namespace std;
class A
{
public:
   int a;
        static int count; // Static Variable

        A()
        {
                this->a=0;
                count++;
        }
        A(int a)
        {
                this->a=a;
                count++;

        }

        void DisplayOutput()
        {
                this->a=100;
                cout<<endl<<"A:::a==="<<this->a<<"\t"<<"Count Is:::"<<count<<endl;
        }

        static void SetCounter(int counter)  // Static Member Function
        {
                A::count=counter;
        }
        ~A()
        {

        }
};

int A::count=100; // Global Definition Of Static Function

int main(void)
{
        A::SetCounter(1000);
        A obj(999);
        obj.DisplayOutput();

        obj.SetCounter(111);
        return 0;
}
```



Size of the Object

Size of the object is size of all non-static data member declared in this class.

We can declare data member as a static. At that time, instead of getting copy of the static variable, all the objects shares single copy of the static data member available in the data segment. That's why size of the static data member is not considered in the size of the object.

If you want to share the value of any data member throughout all the objects, we should declare that data member as static.

Static Member Function

If you want to call any member function without object name, we should declare that member function as a static function. Static member functions are mainly design to call by using class name only but we can call using object also. Static member function can access only static data member.

Static Member Function Doesn’t Have This Pointer

When we call member function of a class by using object implicitly this pointer is passed that member function.

When we call member function of a class by using class name, implicitly this pointer is not passed to that member function.

Static member function is designed to call by class name only. That why this pointer is not passed to static member function of the class.

Member that we can call using object of the class is called instance members.

Member that we can call using class name, these are called as class member.

Member Initializer List

We can initialize either inside constructor body or constructor member initializer list.

When we initialize data member inside constructor member initializer list, increases the efficiency of code.

```
A(int a,int b):a(a),b(b)
{
}
```

Constant Data Member

In C language it is not compulsory to initialize constant variable at the time of declaration because we can modify the value of the constant using pointer.

But in C++, it is compulsory to initialize at the time declaration otherwise compile time error will occur. We can’t initialize data member at the time of declaration. If you want to initialize them, initialize in the constructor member initializer list.

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

```
class A
{
public:
	int a;
	const int b;
	A():a(0),b(0)
	{
		this->a=0;
	}
	A(int a,int b):a(a),b(b)
	{

	}
	void DisplayOutput()
	{
		cout<<endl<<"A:::a==="<<this->a<<"\t"<<this->b<<endl;
	}
	~A()
	{

	}
};


int main(void)
{
	A obj(100,200);
	obj.DisplayOutput();

	return 0;
}
```

When you declare data member as a constant, it is compulsory to initialize that data member inside constructor member initializer list.

Constant Member Function

We can’t declare global function as constant. We can declare member function as constant in C++. When we declare member function as constant, we can’t modify state of object only with that constant member function.

Below program will not compile (C style syntax)

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

 void function() const
{
	cout<<endl<<"Inside const void function()"<<endl;
}

int main()
{
        function();

        return 0;
}
```

Below program will compile and works fine (C++ style syntax)

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

const void function()
{
        cout<<endl<<"Inside const void function()"<<endl;
}

int main()
{
        function();

        return 0;
}
```

We should declare member function as a constant in which we are not modifying state of object.

If you want to modify the state of object inside constant member function at that time declared data member function as mutable.

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

class A
{
public:
        mutable int a;
        const int b;
        A():a(0),b(0)
        {
                this->a=0;
        }
        A(int a,int b):a(a),b(b)
        {

        }
        void DisplayOutput() const
        {
                this->a=100;
                cout<<endl<<"A:::a==="<<this->a<<"\t"<<this->b<<endl;
        }
        ~A()
        {
        }
};


int main(void)
{
        A obj(100,200);
        obj.DisplayOutput();

        return 0;
}
```

Object Oriented Programming Structure (OOPS)

It is a programming methodology to organize complex program into simple program in terms of classes and object. Such methodology is called OOPS.

It is a programing methodology to organize complex program into simple program by using concept of abstraction, encapsulation and inheritance.

Languages which support abstraction, encapsulation, polymorphism and inheritance are called object oriented programming language.

There are four major pillar of OOPS

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

There are there minor pillar of OOPS

1. Polymorphism
2. Concurrency
3. Persistence

Abstraction

Getting only essential thing and hiding unnecessary details is called abstraction.

Example: printf() function

Abstraction always represents the outer behavior of object.

Encapsulation

Binding data and code together is called as encapsulation. Implementation of abstraction is called as encapsulation.

Encapsulations always describe the inner behavior of the object.

Function call is the abstraction and function definition is encapsulation.

Abstraction always changes from user to user.

Information Hiding

Data: unprocessed row material is called data.

Processed data is called information.

Hiding information from user is called information hiding.

In C++ we can use access specifier to provide information hiding.

Modularity

Dividing program into small module for the purpose of simplicity is called as modularity.

There are two types of modularity

1. Physical modularity
2. Logical modularity

Physical Modularity

Diving classes into multiple file is nothing but physical modularity e.g. .h .cpp.

Logical Modularity

Diving classes into namespace is called logical modularity.

Polymorphism

One interface having multiples form is called as polymorphism.

It is two types

1. Compile time polymorphism
2. Run time polymorphism

Synonyms/ different Name of Compile and Runtime Polymorphism

Compile time polymorphism	Runtime polymorphism
Static polymorphism	Dynamic polymorphism
Static binding	Dynamic binding
Weak typing	Strong binding
False polymorphism	True polymorphism

Compile Time Polymorphism

When called to the function resolved at compile time, it is called compile time polymorphism and it is achieved by using function overloading and operator overloading.

Run Time Polymorphism

When called to the function resolved at run time, it is called run time polymorphism and it is achieved by using function overriding.

Hierarchy

Order or level of abstraction is called as hierarchy.

There are three types of hierarchy

1. Has a hierarchy(composition)(has a relationship)
2. Is a hierarchy(inheritance)(is a relationship)
3. Use a hierarchy(dependency)(use a relationship)

Composition

When one object is made from other small's object, it is called as composition.

When object is composed of other object, it is called as composition (nested structure in C programming).

Example: Room has a wall.

System unit has a mother board

```
#include<iostream>
```

```
using namespace std;

class A
{
public:
	int a;
	A()
	{
		this->a=0;
	}
	A(int a)
	{
		this->a=a;
	}
	void DisplayOutput()
	{
		cout<<endl<<"A:::a==="<<this->a<<endl;
	}
	~A()
	{

	}
};

class B
{
public:
	int b;
	A _a;
	B()
	{
		this->b=0;
		this->_a.a=0;
	}
	B(int a,int b)
	{
		this->_a.a=a;
		this->b=b;
	}
	void DisplayOutput()
	{
		cout<<endl<<"B:::a==="<<this->_a.a<<"\t"<<"B::::b===="<<this-
>b<<endl;
	}
	~B()
	{

	}
};

int main(void)
{
	B obj(100,200);
	obj.DisplayOutput();
	return 0;
}
```

Types of Composition

1. Association
2. Aggregation
3. Containment

Association

Removal of a small object does not affect big object, it is called as association.

Example: Room has a chair

Association is having loose coupling

Aggregation

Removal of small object affect big object, it is called as aggregation.

Example: Room has wall

Aggregation is having tight coupling

Containment

Stack, queue, link list, array, and vector these are collectively called collections.

When class contain object of collection is called as containment.

Example: Company has a no of employees.

In composition we always declare one class as a data member into another class.

Inheritance

Acquiring all the states (all the data member) and behavior (member function) of one class (base class) by another class (derived class), such concept is called as inheritance.

At the time of inheritance when the name of the member of the base class and the name of the member of the derived class are same, at that time explicitly mention scope resolution operator is a job of programmer.

When you create object of a derived class at that time constructor of the base class will call first and then the constructor of the derived class will call.

Destructor calling sequence is exactly opposite i.e. destructor of the derived class will call first then destructor of the base class will get call.

At the time of inheritance, all the data member and member function are inherited. But there are some functions which are not inherited into derived class.

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

When we create object of a derived class, at that time size of the object is size of all non-static data member declared in the base class plus size of all non-static data member declared into derived class.

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

class A
{
public:
	int a;
	A()
	{
		this->a=0;
	}
	A(int a)
	{
		this->a=a;
	}
	void DisplayOutput()
	{
		cout<<endl<<"A:::a==="<<this->a<<endl;
	}
	~A()
	{

	}
};

class B:public A
{
public:
	int b;
	B()
	{
		this->b=0;
	}
	B(int a,int b)
	{
		this->a=a;
```

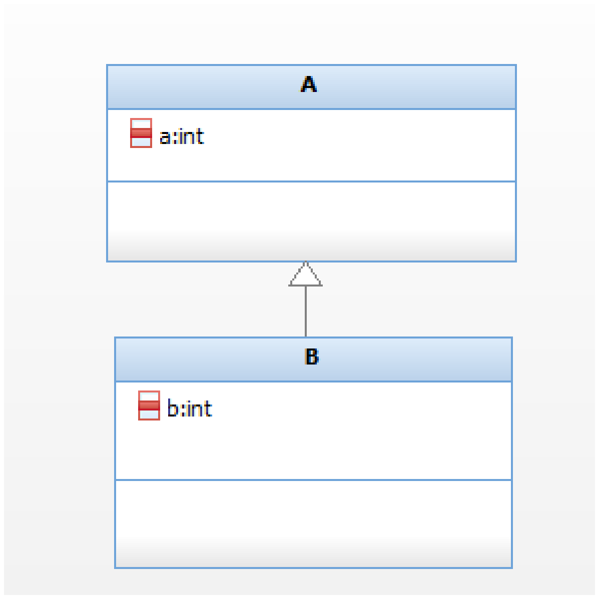
```
			this->b=b;
		}
		void DisplayOutput()
		{
			cout<<endl<<"B:::a==="<<this->a<<"\t"<<"B::::b===="<<this-
>b<<endl;
		}
		~B()
		{
		
		}
};

int main(void)
{
		B obj(100,200);
		obj.DisplayOutput();
		return 0;
}
```

Types of Inheritance

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

Single Inheritance

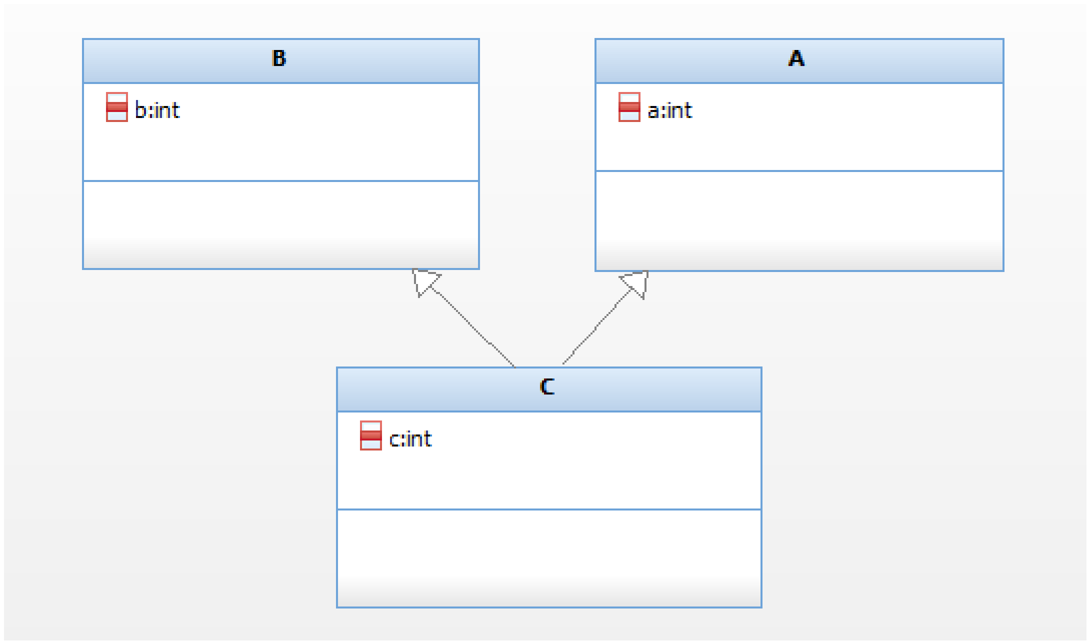
B is derived from A



One class having only one derived class is called as single inheritance.

Multiple Inheritances

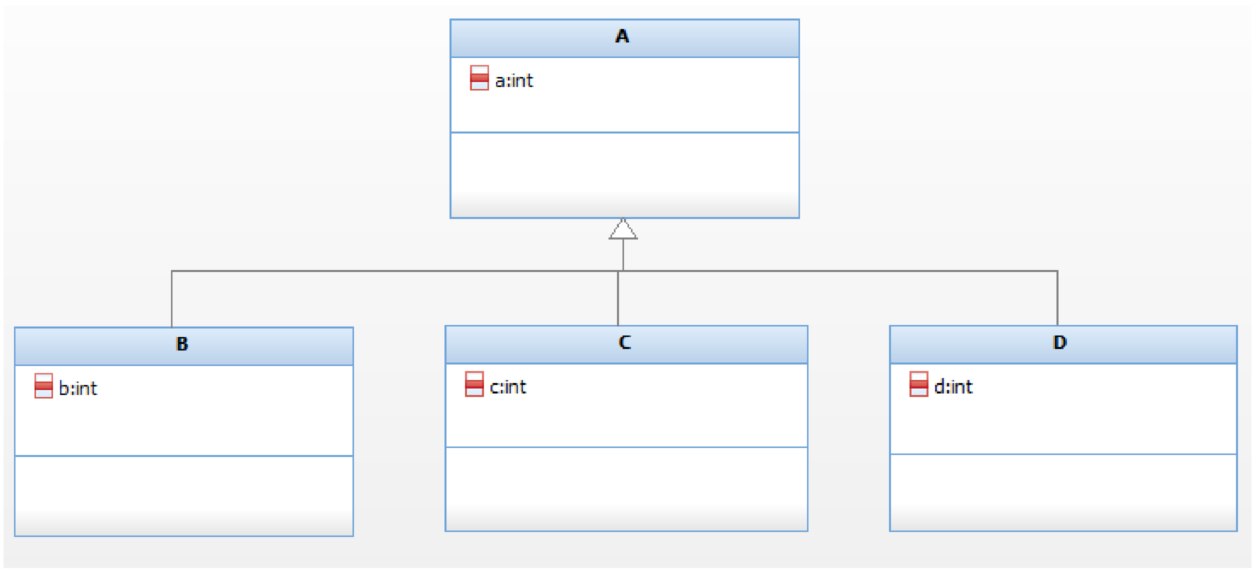
C is derived from class A and class B



Multiple base classes having only one derived class, such type of inheritance is called multiple inheritances.

Hierarchical Inheritance

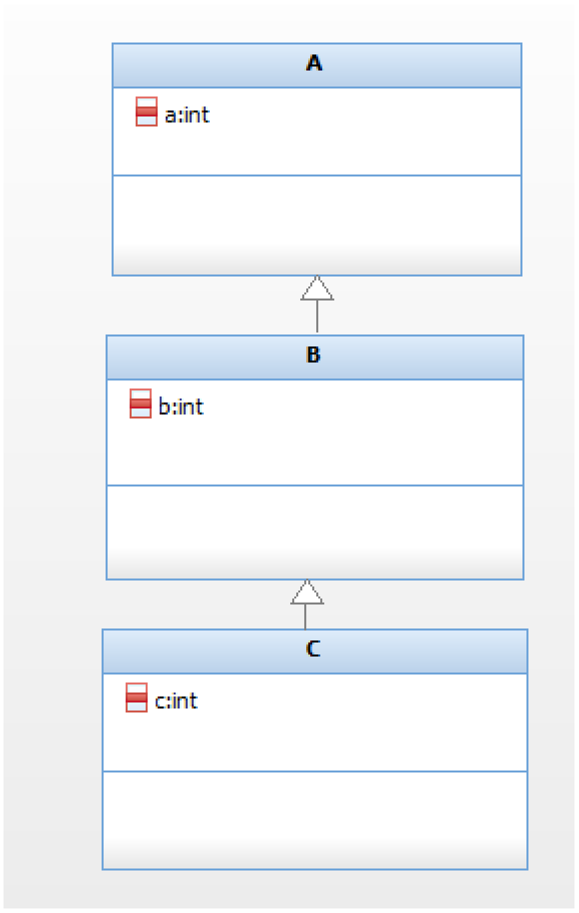
Class B, C and D are inherited from class A.



One base class having multiple derived classes, such type of inheritance is called hierarchical inheritance.

Multilevel Inheritance

Class B is derived from class A and class C is derived from class B.

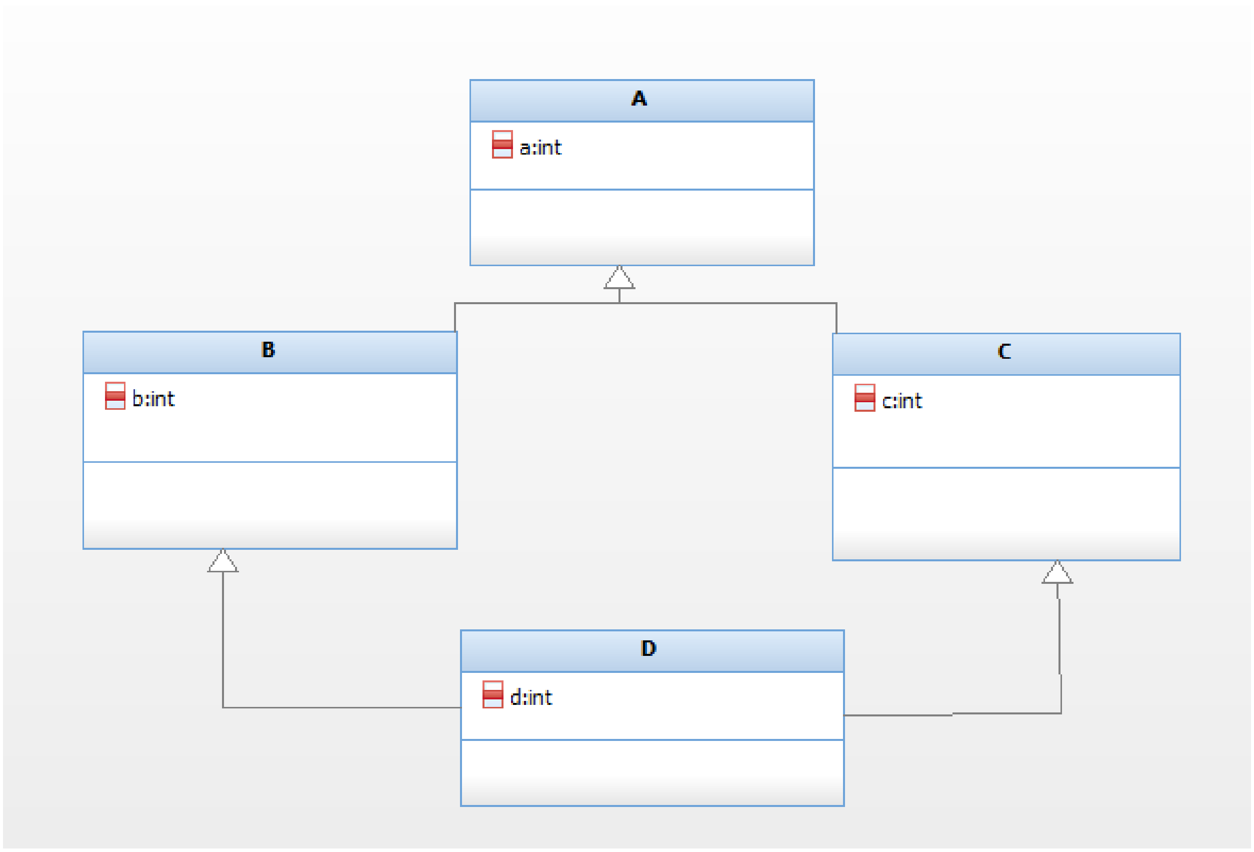


When single inheritance has multi levels, such type of inheritance is called as multilevel inheritance.

Hybrid Inheritance

Combination of all inheritance is called as hybrid inheritance.

Diamond problem or Virtual Base Class or Virtual Inheritance



Constructor calling sequence

A->B->A->C->D

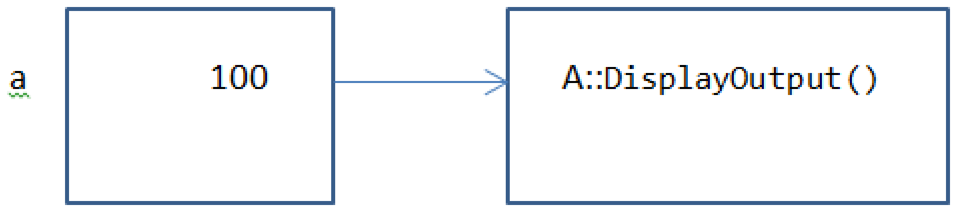
Destructor calling sequence

D->C->A->B->A

Class A is a direct base class for class A and C.

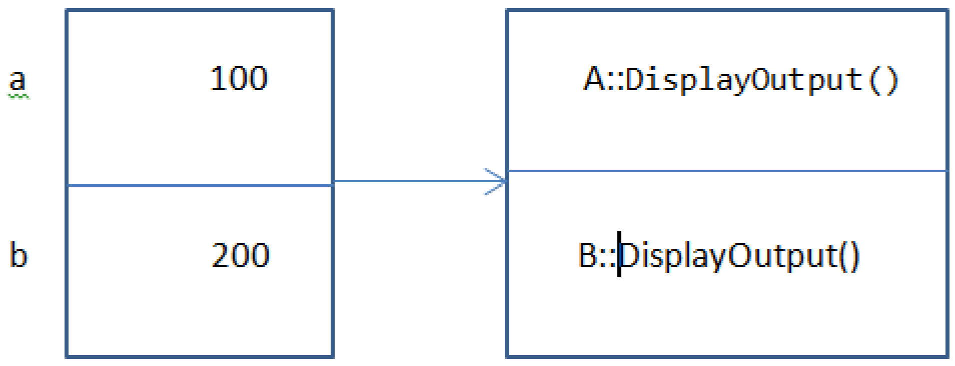
Class B and C are direct base classes for D.

Class A is indirect base class for class D.

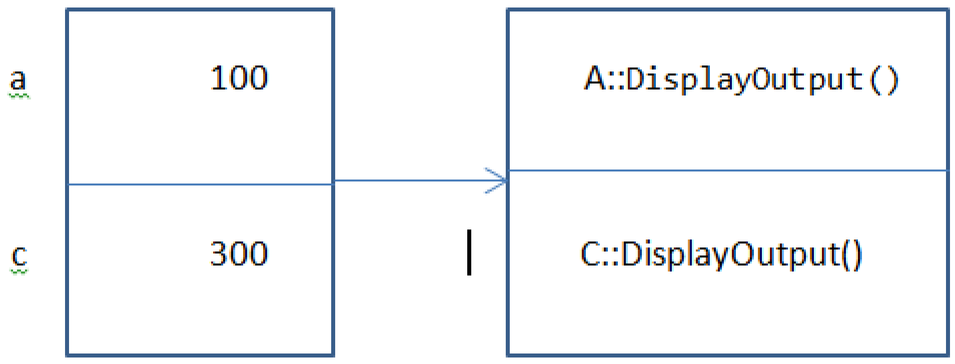


When you create object of class A, it will a single copy of all the data member declared in class A.

Class B is derived from class A. when you create object of B, it will get a single copy of non-static data member declared in class B as well as class A.

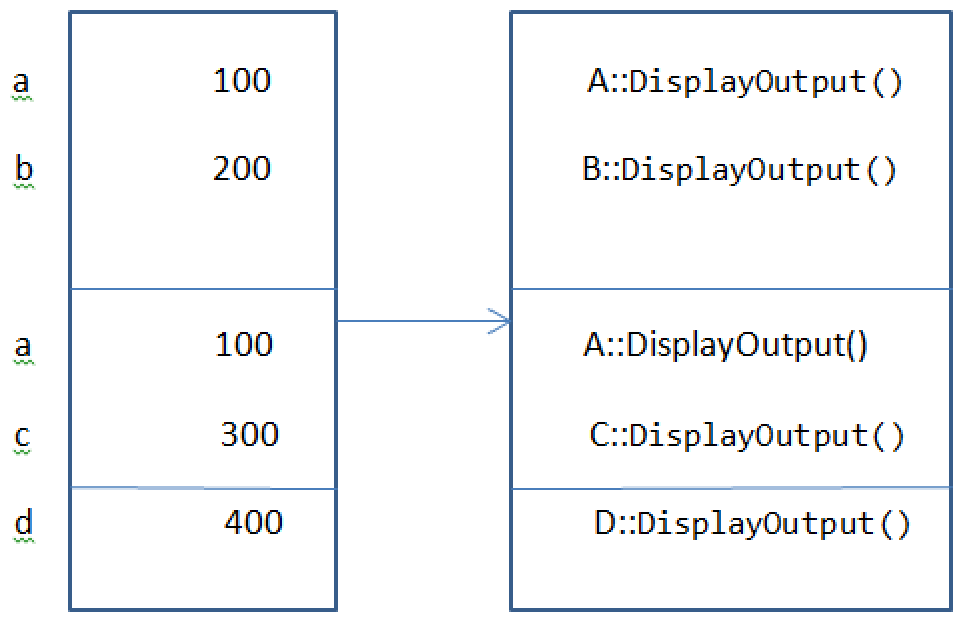


Class C is derived from class A. when you create object of C, it will get a single copy of non-static data member declared in class C as well as class A.



Class D is a derived class which is derived from class B and class C. This is called as multiple inheritances.

When we create object of class D, it will get a single copy of all the data member declared in D as well class B and class C.



In the above diagram class A is a indirect base class for class D. That why all the data member and member function of class A are available twice in class D. when we tried to access data member and member function of class A by using object class D, compiler will confuse(which members to be accessed available from classes A and B). Such problem created by hybrid inheritance is called as diamond problem.

Solution 1 of diamond problem

Explicitly mentioning the name of the class of which data member and member function you want to access.

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

class A
{
public:
	int a;
	A()
	{
		this->a=0;
	}
	A(int a)
	{
		this->a=a;
	}
	void DisplayOutput()
	{
		cout<<endl<<"A:::a==="<<this->a<<endl;
	}
	~A()
	{

	}
};

class B: public A
{
public:
	int b;
	B()
	{
		this->b=0;
	}
	B(int a,int b)
	{
		this->a=a;
		this->b=b;
	}
	void DisplayOutput()
	{
		cout<<endl<<"B:::a==="<<this->a<<"\t"<<"B::::b===="<<this-
>b<<endl;
	}
	~B()
	{

	}
};

class C: public A
{
public:
	int c;
	C()
	{
		this->c=0;
	}
	C(int a,int c)
	{
		this->a=0;
		this->c=0;
	}
	void DisplayOutput()
	{
		cout<<endl<<"C:::a==="<<this->a<<"\t"<<"C:::c===="<<this-
>c<<endl;
```

```
	}
	~C()
	{

	}
};


class D:public B, public C
{
public:
	int d;
	D()
	{
		this->d=0;
	}
	D(int a,int b,int c,int d)
	{
		this->B::a=a;
		this->b=b;
		this->c=c;
		this->d=d;
	}
	void DisplayOutput()
	{
		cout<<endl<<"D:::a==="<<this->B::a<<"\t"<<"D:::b==="<<this-
>b<<"D:::c===="<<this->c<<"\t"<<"D:::d==="<<this->d<<endl;
	}
	~D()
	{

	}
};
int main(void)
{
	D obj(100,200,300,400);
	obj.DisplayOutput();

	obj.B::a=999;

	obj.DisplayOutput();

	return 0;
}
```

Solution 2 of diamond problem

Declare base class as virtual i.e. derived class B from class A virtually and class C from class A virtually.

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

class A
{
```

```
public:
	int a;
	A()
	{
		this->a=0;
	}
	A(int a)
	{
		this->a=a;
	}
	void DisplayOutput()
	{
		cout<<endl<<"A:::a==="<<this->a<<endl;
	}
	~A()
	{

	}
};

class B: virtual public A
{
public:
	int b;
	B()
	{
		this->b=0;
	}
	B(int a,int b)
	{
		this->a=a;
		this->b=b;
	}
	void DisplayOutput()
	{
		cout<<endl<<"B:::a==="<<this->a<<"\t"<<"B::::b===="<<this-
>b<<endl;
	}
	~B()
	{

	}
};

class C: virtual public A
{
public:
	int c;
	C()
	{
		this->c=0;
	}
	C(int a,int c)
	{
		this->a=0;
```

```
                this->c=0;
        }
        void DisplayOutput()
        {
                cout<<endl<<"C:::a==="<<this->a<<"\t"<<"C:::c===="<<this-
>c<<endl;
        }
        ~C()
        {

        }
};


class D:public B, public C
{
public:
        int d;
        D()
        {
                this->d=0;
        }
        D(int a,int b,int c,int d)
        {
                this->a=a;
                this->b=b;
                this->c=c;
                this->d=d;
        }
        void DisplayOutput()
        {
                cout<<endl<<"D:::a==="<<this->a<<"\t"<<"D:::b==="<<this-
>b<<"D:::c===="<<this->c<<"\t"<<"D:::d==="<<this->d<<endl;
        }
        ~D()
        {

        }
};
int main(void)
{
        D obj(100,200,300,400);
        obj.DisplayOutput();

        obj.a=999;

        obj.DisplayOutput();

        return 0;
}
```

Such type of inheritance is called as virtual inheritance.

Now in this case class D will get single copy of all data member and member function of indirect base class A

Mode of Inheritance

When you use private, public and protected at the time of inheritance, it is called as mode of inheritance.

Public Inheritance

When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Protected Inheritance

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

Private Inheritance

When deriving from a private base class, public and protected members of the base class become private members of the derived class

```
Example: class A:public A
```

Here mode of inheritance is public

In C++ by default mode of inheritance is private.

Public Mode of Inheritance

	Base	Derived	Outside class
Private	A	NA	NA
Protected	A	A	NA
Public	A	A	A

Private Mode of Inheritance

	Base	Derived	Outside class
Private	A	NA	NA
Protected	A	A	NA
Public	A	A	NA

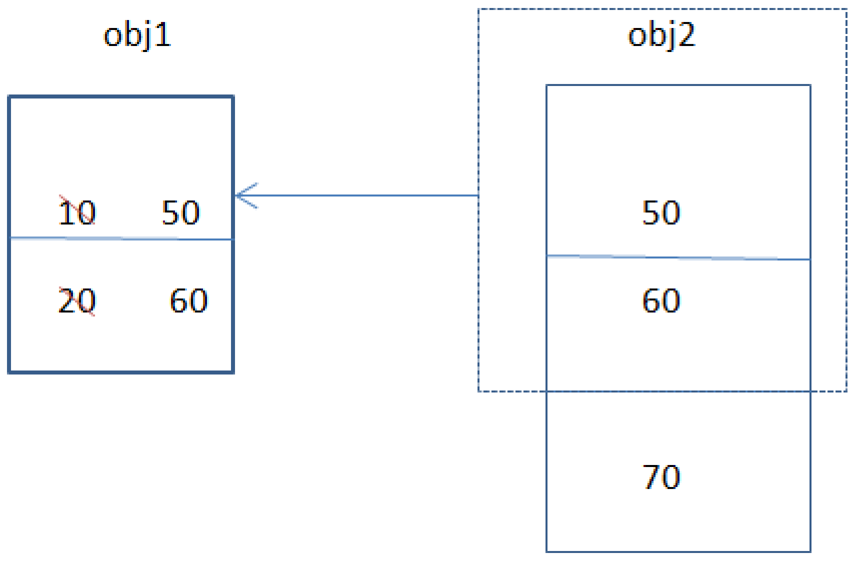
Protected Mode of Inheritance

	Base	Derived	Outside class
Private	A	NA	NA
Protected	A	A	NA
Public	A	A	NA

Object Slicing

When we assign derived class object into base class object, at that time base class portion which is available in derived class is assign to base class object. Such slicing (cutting) of base class portion from the derived class object is called as object slicing.

```
Base obj1(10,20);

Derived obj2(50,60,70);

obj1=obj2 //Object Slicing
```



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

class Base
```

```
{
protected:
        int a,b;

public:
        Base()
        {
                this->a=0;
                this->b=0;
        }

        Base(int a,int b)
        {
                this->a=a;
                this->b=b;
        }

   void DisplayOutput()
        {
                cout<<endl<<"A is:::"<<this->a<<"\t"<<"B is:::"<<this->b<<endl;
        }
        ~Base()
        {

        }
};

class Derived:public Base
{
protected:
        int c;

public:
        Derived()
        {
                this->c=0;
        }
        Derived(int a,int b,int c)
        {
                this->a=a;
                this->b=b;
                this->c=c;
        }
        void DisplayOutput()
        {
                cout<<endl<<"A Is:::"<<this->a<<"\t"<<"B Is:::"<<this->b<<"\t"<<"C
is:::"<<this->c<<endl;
        }
        ~Derived()
        {

        }
};
int main(void)
{
        Derived obj1(100,200,300);
        obj1.DisplayOutput();

        Base obj2=obj1; // Object Slicing
        obj2.DisplayOutput();

        return 0;
}
```

// Draw diagram here

Up Casting

Storing address of derived class object into base class pointer, such concept is called as up casting.

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

class Base
{
protected:
        int a,b;

public:
        Base()
        {
                this->a=0;
                this->b=0;
        }

        Base(int a,int b)
        {
                this->a=a;
                this->b=b;
        }

        virtual void DisplayOutput()
        {
                cout<<endl<<"A is:::"<<this->a<<"\t"<<"B is:::"<<this->b<<endl;
        }
        virtual ~Base()
        {

        }
};

class Derived:public Base
{
protected:
        int c;

public:
        Derived()
        {
                this->c=0;
        }
        Derived(int a,int b,int c)
        {
                this->a=a;
                this->b=b;
                this->c=c;
        }
        void DisplayOutput()
        {
                cout<<endl<<"A Is:::"<<this->a<<"\t"<<"B Is:::"<<this->b<<"\t"<<"C
is:::"<<this->c<<endl;
        }
        ~Derived()
        {

        }
};
int main(void)
{
        Base *ptr=new Derived(100,200,300);
        ptr->DisplayOutput();

        return 0;
}
```

Down Casting

Storing the address of base class object into derived class object is called as down casting.

```
class Base
{
private:
        int num1;
public:
        Base(int num)
        {
                this->num1=num;
        }
        void printoutput(void)
        {
                cout<<endl<<"Inside Base::Number Is:::"<<this->num1<<endl;
        }

        virtual void displayoutput1()
        {
                cout<<endl<<"Inside base class displayoutput1 method"<<endl;
        }

        ~Base()
        {}
};
```

```
class Derived:public Base
{
private:
	int num2;
public:
	Derived(int num):Base(num)
   {
		this->num2=num;
   }

	void printoutput()
	{
		cout<<endl<<"Inside Derived class:::Number is:::"<<this->num2<<endl;
	}

	void displayoutput2()
	{
		cout<<endl<<"Inside derived class displayoutput2 method"<<endl;
	}

	~Derived()
	{}
};


int main(void)
{
	Base *obj1=new Derived(100);

	Derived *ptr=dynamic_cast<Derived*>(obj1);

	ptr->displayoutput1();
	ptr->displayoutput2();
	ptr->printoutput();
	return 0;
}
```

Virtual Function

Function which gets called depending on the type of object rather than the type of pointer, such function is called as virtual function.

Class which contains at least one virtual function, such type of class is called polymorphic class. This is late binding.

Late Binding

When call to the virtual function is given either by using pointer or reference than it is called late binding. In rest of the cases it is early binding.

Function Overriding

Virtual function define in the base class is once again redefine in the derived class is called as function overriding.

Function which takes part into function overriding, such function are called overriding function.

For function overriding function in the base class must be virtual.

Difference between Function Overloading and Function Overriding

Function Overloading	Function Overriding
1. It is compile time polymorphism	1. It is run time polymorphism
2. Signature of the function must be different	2. Signature of the function must be same
3. For function overloading, function must be same scope i.e. either function must be global or it must be inside the same class only.	3. Function must be in base and derived class
4. Function gets call by looking toward the mangled name	4. Function gets called by looking towards vtable.

Virtual Function Table

When class contains at least one virtual function at that time compiler internally creates one table which stores the address of virtual function. Such table is called virtual function table or vtable.

Virtual Function Pointer

When class contains virtual function, internally vtable is created by the compiler and to store the address of vtable, compilers implicitly add one hidden member inside a class which store the address of vtable. Such hidden member is called as virtual function pointer or vfptr or vptr.

Pure Virtual Function

Virtual function which is equated to zero, such virtual function is called as pure virtual function.

Generally pure virtual function doesn’t have body.

Class which contains at least one pure virtual function is known as abstract class.

If class is abstract we can’t create object of that class but we can create pointer or reference of that class.

It is not compulsory to override virtual function but it is compulsory to override pure virtual function.

If don’t override pure virtual function in the derived class at that time derived class can be treated as abstract class.

Abstract class can have non virtual function, virtual function as well as pure virtual function.

If you are able to create object of your class is called as concrete class.

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

class Base
{
protected:
        int a,b;

public:
        Base()
        {
                this->a=0;
                this->b=0;
        }

        Base(int a,int b)
        {
                this->a=a;
                this->b=b;
        }

   virtual void DisplayOutput()=0;

        virtual ~Base()
        {

        }
};

class Derived:public Base
```

```
{
protected:
	int c;

public:
	Derived()
	{
		this->c=0;
	}
	Derived(int a,int b,int c)
	{
		this->a=a;
		this->b=b;
		this->c=c;
	}
	void DisplayOutput()
	{
		cout<<endl<<"A Is:::"<<this->a<<"\t"<<"B Is:::"<<this->b<<"\t"<<"C
is:::"<<this->c<<endl;
	}
	~Derived()
	{

	}
};
int main(void)
{
	Base *ptr=new Derived(100,200,300);
	ptr->DisplayOutput();

	return 0;
}
```

Template

Swapping of data of any data type

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

template<class T>

void SWAP(T &num1, T &num2)
{
	T temp;
	temp=num1;
	num1=num2;
	num2=temp;
}// Template SWAP Function

int main(void)
{
	int a=100,b=200;
	cout<<endl<<"==========Before Swapping============"<<endl;
```

```
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        SWAP(a,b);

        cout<<endl<<"==========After Swapping============"<<endl;
        cout<<endl<<"A Is:::"<<a<<"\t"<<"B Is:::"<<b<<endl;

        return 0;
}
```

```
Syntax : template <class T>

         template <typename T>
```

Generic Programming Template

Many times we need to write the code that is common for many data type.

Example: swap function

We observed that logic of swapping remain same irrespective of data type used. We can write template of the function instead of writing separate function for each data type. The syntax is as bellows

```
template<class T>

void SWAP(T &num1, T &num2)
{
        T temp;
        temp=num1;
        num1=num2;
        num2=temp;
}// Template SWAP Function
```

Now the same function can be used to swap two integer variable or float variable or even any user defined data types variable.

If the function is called as

```
SWAP(a,b)
```

Where a and b are integer variable, compiler convert the above template function to the function for swapping int variables. To do this compiler simply converts above function to new function replacing ‘T’ with ‘int’. Thus compiler creates the functions corresponding to every data type for which function has been called.

Template classes are also called Meta classes because the compiler generates real classes from template classes and then create the object.

Sample Template Class Program

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

#define SIZE 5

template<class T>

class Tarray
{
private:
	T _array[SIZE];
public:
	Tarray()
	{
		for(int i=0;i<SIZE;i++)
		{
			this->_array[i]=i;
		}
	}

	void AcceptInput()
	{
		cout<<endl<<"Enter the Array content:::"<<endl;
		for(int i=0;i<SIZE;i++)
		{
			cin>>this->_array[i];
		}
	}

	void DisplayOutput()
	{
		cout<<endl<<"Entered Array content Is::::"<<endl;
		for(int i=0;i<SZIE;i++)
		{
			cout<<this->_array[i]<<"\t";

		}
	}
	~Tarray()
	{

	}
};

int main(void)
{
	cout<<endl<<"=====INTEGER OBJECT======"<<endl;
	Tarray<int> obj;
	obj.AcceptInput();
	obj.DisplayOutput();

	return 0;
}
```

Casting Operator

1. Static cast
2. Dynamic cast
3. Const cast
4. Reinterpret cast

Static Cast

If you want to use C style casting at that time we should use static_cast operator.

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

int main(void)
{

	double a=200.4;
	cout<<endl<<"Double Number Is::::"<<endl;

	int b=static_cast<int>(a);
	cout<<endl<<"Integer Number Is:::"<<b<<endl;

	return 0;
}
```

Reinterpret Cast

If you want cast any type into another incompatible type at that time we should use reinterpret_cast.

Note: How to access private data member of class outside classes (use reinterpret_cast)

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

class Base
{
private:
	int a,b;

public:
	Base()
	{
		this->a=0;
		this->b=0;
	}

	Base(int a,int b)
	{
		this->a=a;
```

```
                this->b=b;
        }

        void DisplayOutput()
        {
                cout<<endl<<"A is:::"<<this->a<<"\t"<<"B is:::"<<this->b<<endl;
        }
        ~Base()
        {

        }
};

int main(void)
{

        Base obj(100,200);
        obj.DisplayOutput();

        int *ptr=reinterpret_cast<int *>(&obj);
        cout<<endl<<"A Is:::"<<*ptr<<endl;
        ptr++;
        cout<<endl<<"B is:::"<<*ptr<<endl;

        return 0;
}
```

Const Cast

This casting operator is used to remove the constness of any pointer temporary.

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

void function(const int *num)
{
        int *temp=const_cast<int *>(num);
        *temp=500;
}

int main(void)
{

        int a=100;
        cout<<endl<<"Before Modification Number Is:::"<<a<<endl;

        function(&a);

        cout<<endl<<"After Modification Number Is:::::"<<a<<endl;


        return 0;
}
```

Dynamic Cast

This casting operator verifies the validity of cast at runtime. If it is valid cast success else cast fails. If cast fails in case of pointer then it returns NULL pointer and it is fails in case of reference, it throws bad_cast exception. Dynamic cast internally uses RTI and hence class must have minimum one virtual function in the class. In other words class must be polymorphic class otherwise compile time error.

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

class Base
{
protected:
        int a,b;

public:
        Base()
        {
                this->a=0;
                this->b=0;
        }

        Base(int a,int b)
        {
                this->a=a;
                this->b=b;
        }

        virtual void DisplayOutput()
        {
                cout<<endl<<"A is:::"<<this->a<<"\t"<<"B is:::"<<this->b<<endl;
        }
        virtual ~Base()
        {

        }
};

class Derived:public Base
{
protected:
        int c;

public:
        Derived()
        {
                this->c=0;
        }
        Derived(int a,int b,int c)
        {
                this->a=a;
                this->b=b;
                this->c=c;
```

```
	}
	void DisplayOutput()
	{
		cout<<endl<<"A Is:::"<<this->a<<"\t"<<"B Is:::"<<this->b<<"\t"<<"C 
is:::"<<this->c<<endl;
	}
	~Derived()
	{

	}
};
int main(void)
{

	Derived obj(100,200,300);
	obj.DisplayOutput();

	Base *ptr=dynamic_cast<Base *>(&obj);
	ptr->DisplayOutput();

	return 0;
}
```

Namespace

Namespaces are used to localize the identifier to avoid name collisions. Name collision may occur for global variables, global function, classes etc. It is necessary when two or more third party libraries are used by the same program because a name define by one library may conflict with name defined by other library.

Default namespace is global namespace and can access global data and function by using scope resolution operator.

For example :: a gives access to global data variable ‘a’. We can create our own namespace and anything define within namespace has scope limited.

As per ANSCII/ISO standard of C++ use of namespace is compulsory. All the predefine function and classes are defined with in the standard called ‘std’.

A simple C++ can be written as follows

```
#include<iostream>

int main(void)
{
	std::cout<<std::endl<<"Hello World C++ Program"<<std::endl;
	return 0;
}
```

This can be written for avoiding repeated use of ‘std’ using namespace as follows

```
#include<iostream>
```

```
using namespace std;

int main(void)
{
	cout<<endl<<"Hello World C++ Program"<<endl;
	return 0;
}
```

Sample Namespace Program

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

int num=100;
namespace NA
{
	int num=200;
}
int main(void)
{
	int num=300;

	cout<<endl<<"Local Variable Num is::::"<<num<<endl;
	cout<<endl<<"NA::num Is:::"<<NA::num<<endl;
	cout<<endl<<"Global Num is::"<<::num<<endl;


	return 0;
}
```

Sample Namespace Program Using namespace

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

namespace NA
{
	int num=200;
}
int main(void)
{

	using namespace NA;

	cout<<endl<<"NA::num Is:::"<<num<<endl;

	return 0;
}
```

Exception Handling

Exceptions are certain situation which may occur in the program causing failure of the program. Typical examples are diving by zero, array index outside bound.

C++ provided power full method of handling such exception using try, catch and throw keywords.

Try block is always looking for an exception. If try is raising any kind of exception, at that time it is a job of catch block to catch the exception. We have to use throw keywords to explicitly raise the exception. One try block may have multiple catch block but there must be at least one catch block to catch the exception otherwise compiler time error will occur.

Generic Catch Block

Catch block which can handle all kind of exception, such catch block is called as generic catch block. Generic catch block must be the last catch block.

When exception is throw but no matching catch block is not available, at that time compiler implicitly give call to the one library function terminate and terminate implicitly give call to the abort function.

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

int main(void)
{
        int num1=100, num2=0;

        try
        {
                if(num2==0)
                {
                        cout<<endl<<"num2 can't be zero"<<endl;
                        throw num2;
                }
        }

        catch(int a) // Integer Catch Block
        {
                cout<<endl<<"Catch Exception"<<endl;
        }

        catch(...)// Generic Catch Block
        {
                cout<<endl<<"Catch the Exception"<<endl;
        }

        return 0;
}
```

Run Time Type Information

Run Time Type Information (RTTI) is the concept of determining the type of any variable during execution (runtime)

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

int main(void)
{
        int num=100;
        type_info const &other=typeid(num);

        cout<<endl<<"Type Is:::"<<other.name()<<endl;
        return 0;
}
```

Smart Pointer

Smart pointers are objects which store pointers to dynamically allocated (heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners.

Conceptually, smart pointers are seen as owning the object pointed to and thus responsible for deletion of the object when it is no longer needed.

```
int main(void)
{
        Treal *ptr=new Treal(100,200);
        ptr->DisplayOutput();

        return 0;
}
```

Since we forget to delete ptr using delete operator, ptr pointing to the memory is leakage. To avoid such leakage we have to use smart pointer.

Smart Pointer Types

boost::scoped_ptr or std::unique_ptr	Simple sole ownership of single objects. Non copy able.
boost::shared_ptr or std::shared_ptr	Object ownership shared among multiple pointers.

std::auto_ptr	It is very much like a scoped pointer, except that it also has the "special" dangerous ability to be copied — which also unexpectedly transfers
boost::weak_ptr or std:: weak_ptr	Non-owning observers of an object owned by shared_ptr.

Sample program using boost::scoped_ptr

```
#include<iostream>
#include<boost/scoped_ptr.hpp>
using namespace std;
using namespace boost;

class Treal
{
public:
	int real,img;
	Treal()
	{
		this->real=0;
		this->img=0;
		cout<<endl<<"Inside Parameterless constructor"<<endl;

	}

	Treal(int real,int img)
	{
		this->real=real;
		this->img=img;
		cout<<endl<<"Inside parameterised constructor"<<endl;
	}

	void DisplayOutput(void)
	{
		cout<<endl<<"Real No Is:::"<<this->real<<"\t"<<"Img No Is::::"<<this-
>img<<endl;
	}

	~Treal()
	{
		cout<<endl<<"Inside Destructor method"<<endl;
	}
};


int main(void)
{

	scoped_ptr<Treal> ptr(new Treal(100,200));
	ptr->DisplayOutput();
	//scoped_ptr<Treal> ptr2(ptr);  // ptr conn't be copied into ptr2
	//ptr2->DisplayOutput();

	return 0;
}
```

Sample program using boost::shared_ptr

```
#include<iostream>
#include<boost/shared_ptr.hpp>
using namespace std;
using namespace boost;

class Treal
{
public:
	int real,img;
	Treal()
	{
		this->real=0;
		this->img=0;
		cout<<endl<<"Inside Parameterless constructor"<<endl;

	}

	Treal(int real,int img)
	{
		this->real=real;
		this->img=img;
		cout<<endl<<"Inside parameterised constructor"<<endl;
	}

	void DisplayOutput(void)
	{
		cout<<endl<<"Real No Is:::"<<this->real<<"\t"<<"Img No Is::::"<<this-
>img<<endl;
	}

	~Treal()
	{
		cout<<endl<<"Inside Destructor method"<<endl;
	}
};




int main(void)
{
	shared_ptr<Treal> ptr1(new Treal(100,200));
	ptr1->DisplayOutput();

	shared_ptr<Treal> ptr2(ptr1);
	ptr2->DisplayOutput();
```

```
        return 0;
}
```

Sample Program using std::auto_ptr

```
std::auto_ptr<Treal> p1 (new Treal(100,200));

std::auto_ptr<Treal> p2 = p1; // Copy and transfer ownership. p1 gets set to empty!

p2-> DisplayOutput (); // Works.

p1-> DisplayOutput (); // Oh oh. Hopefully raises some NULL pointer exception.
```

Note: auto_ptr is deprecated in the newest standards, so you shouldn't use it. Use the std::unique_ptr instead

Does C++ class create padding similar to structure?

Yes C++ class creates padding similar to structure.

To avoid padding in C++ class, use pragma pack preprocessor.

```
#pragma pack(1)
```

Does pure virtual function has entry in virtual function table

The declaration is required because you need to tell the compiler to reserve a slot in the vtable for that specific method starting from the base class in which it is declared (which is the type you could want to use when calling a method on a derived class)

Just to give you the idea let's make an example (which is not to be considered exactly what happens under the hood). Let's suppose you have three virtual methods in Base, one of which is pure,

```
class Base {

  virtual void pure() = 0;

  virtual void nonpure() { }

  virtual void nonpure2() { }

};
```

so the Base vtable will look like

```
0 [ pure ] -> nothing

1 [ nonpure ] -> address of Base::nonpure

2 [ nonpure2] -> address of Base::nonpure2
```

now let's derive it with

```
class Derive : public Base {

  virtual pure() override { }

  virtual nonpure2() override { }

};
```

Derived vtable will look like

```
0 [ pure ] -> address of Derived::pure

1 [ nonpure ] -> address of Base::nonpure

2 [ nonpure2 ] -> address of Derived::nonpure2
```

When you then try to do

```
Base* derived = new Derived();

derived->pure();

The method is roughly compiled as

address = derived->vtable[0];

call address
```

If you don't declare the pure virtual method in the Base class there is no way to know it's index in the vtable (0) in this case at compile time since the method is not present at all.

But being a hole in the vtable you can't instantiate a class that has a vtable with an empty "slot.

Virtual Function Pointer and Virtual Function Table

This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class.

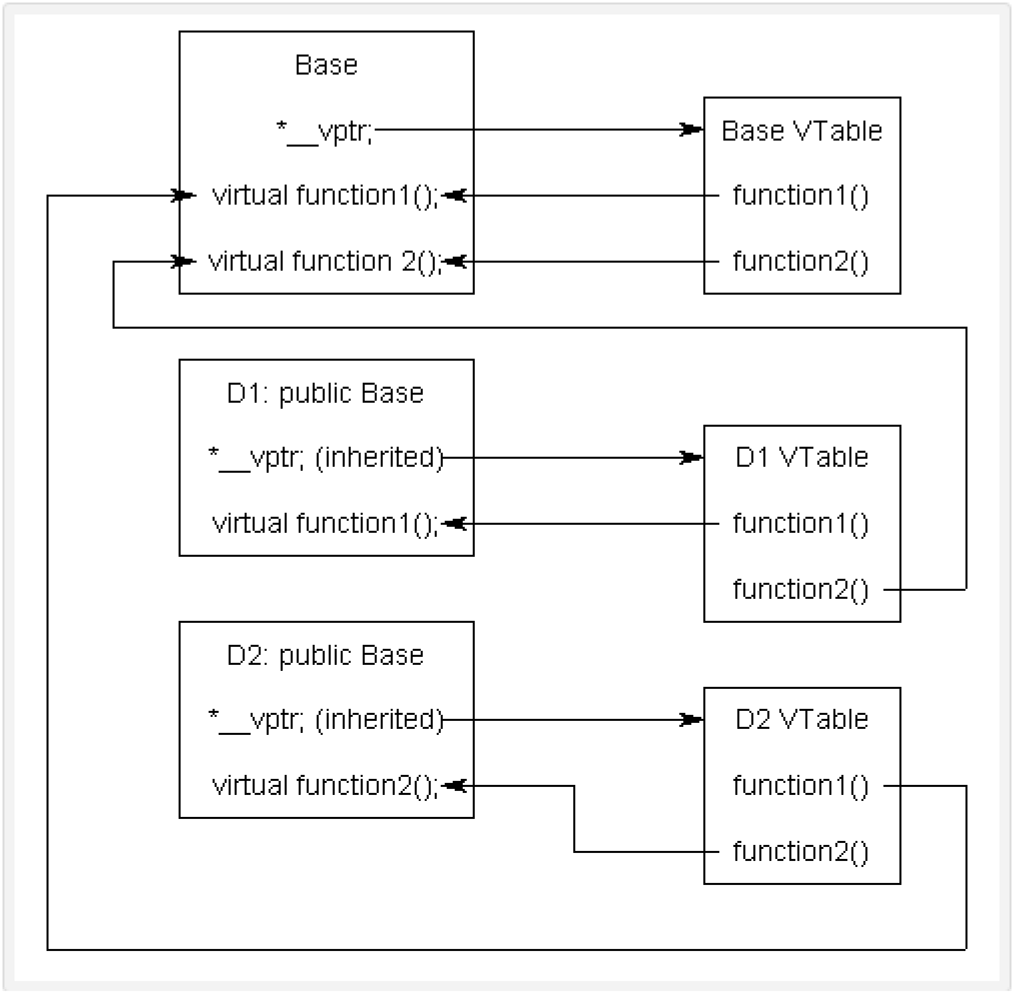
```
class Base
{
public:
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   virtual void function1() {};
};

class D2: public Base
{
public:
   virtual void function2() {};
};
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we'll put it in the next example just to show where it's added:

```
class Base
{
public:
   FunctionPointer *__vptr;
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   virtual void function1() {};
};

class D2: public Base
{
public:
   virtual void function2() {};
};
```

When a class object is created, *__vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, *__vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.



Constant object

In C++, we can create constant object. If you create constant object, at that time you will able to access constant member function only.

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

class Treal
{
```

```
public:
	int real,img;
public:
	Treal()
	{
		this->real=0;
		this->img=0;
	}

	Treal(int real,int img)
	{
		this->real=real;
		this->img=img;
	}

	void DisplayOutput()
	{
		cout<<endl<<"Real No Is:::"<<this->real<<"\t"<<"Img No Is:::"<<this-
>img<<endl;
	}

	inline int getReal()const
	{
		return this->real;
	}

	inline int getImg()const
	{
		return this->img;
	}
};


int main(void)
{

	const Treal obj(100,200);

	cout<<endl<<"Real No Is::::"<<obj.getReal()<<endl;
	cout<<endl<<"Img No Is:::::"<<obj.getImg()<<endl;


	//Treal::DisplayOutput(); // Non constant member function will not be accessible using
constant object

	return 0;
}
```

Do not throw an exception from a destructor

The 2011 C++ Language Standard states that unless a user provided destructor has an explicit exception specification, one will be added implicitly, matching the one that an implicit destructor for the type would have received.

Below program will fail at run time.

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

class Sample
{
public:
        Sample()
   {
                cout<<endl<<"Inside constructor method"<<endl;
   }

        ~Sample()
        {
                throw 100;

        }

};

int main()
{
         Sample obj;

        return 0;
}
```

Furthermore when an exception is thrown, stack unwinding will call the destructors of all objects with automatic storage duration still in scope up to the location where the exception is eventually caught. The program will immediately terminate should another exception be thrown from a destructor of one of these objects

Below program will works fine.

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

class Sample
{
public:
        Sample()
   {
                cout<<endl<<"Inside constructor method"<<endl;
   }
```

```
        ~Sample()
        {
                try
                {
                   throw 100;

                }
                catch(...)
                {

                }
                cout<<endl<<"Inside constructor method"<<endl;
        }

};

int main()
{
         Sample obj;

        return 0;
}
```

Can I throw an exception from a constructor? From a destructor?

For constructors, yes: You should throw an exception from a constructor whenever you cannot properly initialize (construct) an object. There is no really satisfactory alternative to exiting a constructor by a throw.

For destructors, not really: You can throw an exception in a destructor, but that exception must not leave the destructor; if a destructor exits by emitting an exception, all kinds of bad things are likely to happen because the basic rules of the standard library and the language itself will be violated. Don’t do it.