*******************************C++**

Class: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

Can a C++ class have an object of self type:
A class declaration can contain static object of self type, it can also have pointer to self type, but it cannot have a non-static object of self type.
class Test
{
       static Test self; // works fine: Static variables do not contribute to the size of objects. So no problem in calculating size with static variables   of self type.
};
class Test
{
       Test * self; //works fine: For a compiler, all pointers have a fixed size irrespective of the data type they are pointing to, so no problem with this also.
};
class Test
{
       Test self; // Error: If a non-static object is member then declaration of class is incomplete and compiler has no way to find out size of the objects of the class.
} ;
_____

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

Private: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class.

Protected: Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of it's class unless with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass(derived class) of that class as well.

Ex:
class myClass
{
       public:
       int x;
       private:
       int y;
       protected:
       int z;

       void update()
       {
              y = 20; **//private data member can be accessed only inside public data member functions.**
              Z = 30;
       }

};

class subClass: public myclass
{
       public:
       void update_subclass()
       {
              z = 15; **//protected data member can be accessed inside data member function of derived class**

       }
};

int main()
{
       myClass m;
       m.x = 10; **//public data member can be accessed outside the class also and can be directly accessed using obj.datamember.**
       m.update(); //

}

----

4 pillars of oops: Encapsulation, Abstraction, Polymorphism, Inheritance

Encapsulation: is a process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class. It also used for SRP (single responsibility principle). Ex: for game, game rendering can be handled by one class and players can be handled using other class.

1) Make all the data members private.
2) Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member

```cpp
// c++ program to explain
// Encapsulation

#include<iostream>
using namespace std;

class Encapsulation
{
        private:
                // data hidden from outside world
                int x;

        public:
                // function to set value of
                // variable x
                void set(int a)
                {
                        x =a;
                }

                // function to return value of
                // variable x
                int get()
                {
                        return x;
                }
};

// main function
int main()
{
        Encapsulation obj;

        obj.set(5);

        cout<<obj.get();
        return 0;
}
```

Abstraction: is used for directly using the functionalities without knowing the implementation.
Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Data abstraction: hide data using access specifier
Class abstraction: hide implementation.

Inheritance--> It is concept of reusing fictionalities. Should be used only for two cases:

1) Want to reuse complete code/functionality of class & new functionality to be implemented)
   Writing car, truck class but some functionalities will be common can be written in Main class Vehicle and can be inherited.
   ```cpp
   class Vehicle
   {
           Public:
           IsBreakApplied();
           getFuelValue();
           capacity();
   };
   class car: public Vehicle
   class truck: public Vehicle
   ```

2) Want to change/override the functionalities of Base class.

Single level inheritance:
class A

```cpp
{
        int x;
        public:
         A(int x):x(x){}
         int getvalue()
         {
                 cout<<x<<endl;
         }
};

class B:public A
{
        int y;
        public:
         B(int x, int y):A(x),y(y){} --Assigning value to A's class member x using A(x).
};

int main()
{
        B b1(30, 20);
        b1.getvalue();
}
```

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**. **The constructors of inherited classes are called in the same order in which they are inherited.**

```cpp
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};

// second base class
class FourWheeler {
public:
        FourWheeler()
        {
        cout << "This is a 4 wheeler Vehicle" << endl;
        }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
        // creating object of sub class will
        // invoke the constructor of base classes
        Car obj;
        return 0;
}
```
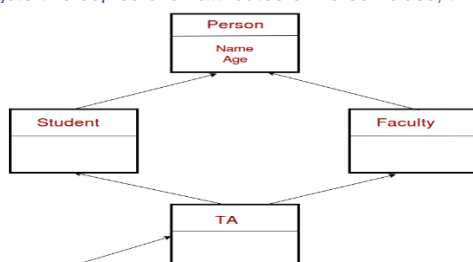
The diamond problem:

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.


```cpp
#include<iostream>
```

```
using namespace std;
class Person {
// Data members of person
public:
        Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : public Person {
// data members of Faculty
public:
        Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
        }
};

class Student : public Person {
// data members of Student
public:
        Student(int x):Person(x) {
                cout<<"Student::Student(int ) called"<< endl;
        }
};

class TA : public Faculty, public Student {
public:
        TA(int x):Student(x), Faculty(x) {
                cout<<"TA::TA(int ) called"<< endl;
        }
};

int main() {
        TA ta1(30);
}
```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. *The solution to this problem is 'virtual' keyword*. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

Solution using virtual keyword:
```
#include<iostream>
using namespace std;
class Person {
public:
        Person(int x) { cout << "Person::Person(int ) called" << endl; }
        Person()   { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
        Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
        }
};

class Student : virtual public Person {
public:
        Student(int x):Person(x) {
                cout<<"Student::Student(int ) called"<< endl;
        }
};

class TA : public Faculty, public Student {
public:
        TA(int x):Student(x), Faculty(x) {
                cout<<"TA::TA(int ) called"<< endl;
        }
};

int main() {
        TA ta1(30);
}
```
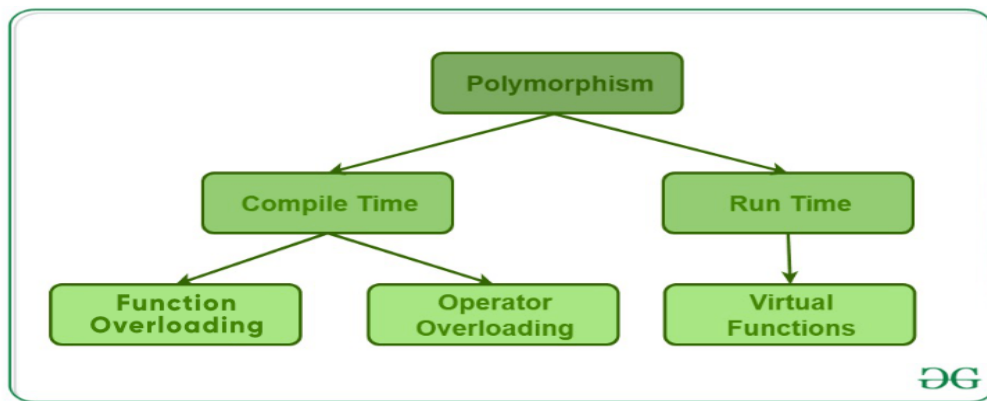
Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.
----

size of class: 1 byte (minimum memory is assigned), to make them distinguishable.

Polymorphism: Many forms. One function can behave differently in different situations.
Compile time: function overloading, operator overloading
Run time: Virtual functions

Name mangling:
C++ compiler distinguishes between different functions when it generates object code – it changes names by adding information about arguments.
int f (void) { return 1; }
int f (int) { return 0; }
void g (void) { int i = f(), j = f(0); }
Compiler does name mangling:
int __f_v (void) { return 1; }
int __f_i (int) { return 0; }
void __g_v (void) { int i = __f_v(), j = __f_i(0); }

Function Overloading:
Two or more functions can have the same name but different parameters/signatures.

When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading "Function" name should be the same and the arguments should be different.

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

void print(int i)
{
        cout << " Here is int " << i << endl;
}
void print(double f)
{
        cout << " Here is float " << f << endl;
}
void print(char const *c)
{
        cout << " Here is char* " << c << endl;
}

int main() {
print(10);
print(10.10);
print("ten");
return 0;
}
```

Functions can not be overloaded if they differ only in the return type

```
int foo() {
return 10;
}

char foo() {
return 'a';
}

int main()
{
char x = foo();
getchar();
return 0;
```

```
}
```

Function overloading and const keyword:
C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a
pointer. That is why the program 1 failed in compilation, but the program 2 worked fine.
Prog1:
```
void fun(const int i)
{
    cout << "fun(const int) called ";
}
void fun(int i)
{
    cout << "fun(int ) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```
Compiler Error: redefinition of 'void fun(int)'

// PROGRAM 2 (Compiles and runs fine)

```
void fun(char *a)
{
cout << "non-const fun() " << a;
}

void fun(const char *a)
{
cout << "const fun() " << a;
}

int main()
{
const char *ptr = "GeeksforGeeks";
fun(ptr);
return 0;
}
```

----

operator overloading: C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope,
which is called **function overloading** and **operator overloading** respectively.
```
#include<iostream>
using namespace std;

class A
{
        int x;
        public:
        A():x(0){}
        A(int x):x(x){}
        A& operator=(A& a1)
        {
                this->x = a1.x;
                return *this;
        }
        //void operator+(int y)
        //{
        //        x= x+y;
//
//        }
        A operator+(int y)
        {
                int z = x+y;
                return A(z);
        }
        int getval()
        {
                cout<<"val is"<<this->x<<endl;
        }
        friend ostream& operator<<(ostream& os, const A& a)
        {
                os<<a.x;
                return os;
        }
```

```
        friend istream& operator>>(istream& is, const A& a)
        {
                is>>a.x;
                return is;
        }


};

int main()
{
        A a1(10);
        A a2, a3;
        a1.getval();
        a2 = a1; //a2.operator=(const A& a1)
        a2.getval();
        //a1+5; // a1.operator+(int)
        //a1.getval();
        a3.getval();
        cout<<a1;
        return 0;
}
```

//New operator and delete operator overloading:

```
#include <cstdio>

#include <cstdlib>

#include <new>

// replacement of a minimal set of functions:

void* operator new(std::size_t sz) {

    std::printf("global op new called, size = %zu\n", sz);

    void *ptr = std::malloc(sz);

    if (ptr)

        return ptr;

    else

        throw std::bad_alloc{};

}

void operator delete(void* ptr) noexcept

{

    std::puts("global op delete called");

    std::free(ptr);

}

int main() {

    int* p1 = new int;

    delete p1;


    int* p2 = new int[10]; // guaranteed to call the replacement in C++11

    delete[] p2;

}
```

Function overriding: is redefinition of base class function in its derived class with same signature i.e return type and parameters.
```
class Base {
    public:
     void print() {
         cout << "Base Function" << endl;
     }
};

class Derived : public Base {
    public:
```

```
        void print() {
            cout << "Derived Function" << endl;
        }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

----

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

Rules for Virtual Functions

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

Problem without Virtual function:
```
class A
{       public:
        void fun1()
        {
                cout<<"A::fun1 is called"<<endl;
        }
};
class B: public A
{
        public:
        void fun1()
        {
                cout<<"B::fun1 is called"<<endl;
        }
        void fun2()
        {
        cout<<"B::fun2 is called"<<endl;
        }
};

int main()
{
        A*p, a1;
        B b1;
        p = &b1; //This will be executed at run time. compiler does not know if p is holding address of object of A/B.
        b1.fun1(); //EB: compile time binding.
        p->fun1(); //EB: compile time binding. compiler will check type of p which is of class A type. It will call class A
function. Ideally we want to call function of class B, since we passed object of class B.
        return 0;
}
//Virtual function concept can be implemented only using pointers. Because pointer may hold address of
//object of any class. Which is evaluated at run time. If we declare Class A's function to virtual.
//Compiler will not do early binding. It will be then late binding.
```
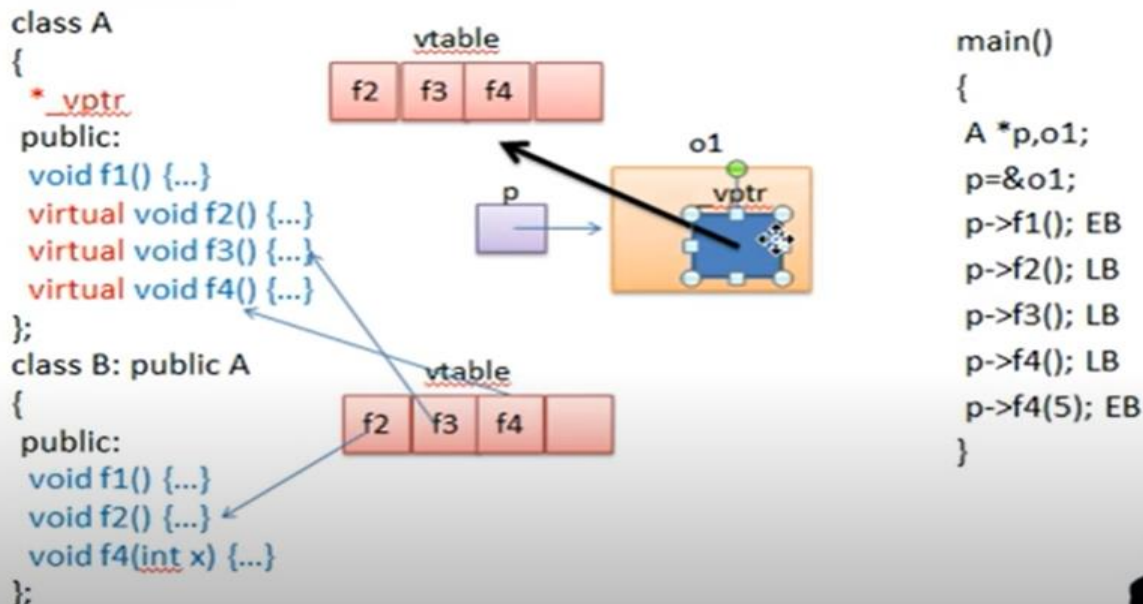
Working of virtual functions (concept of VTABLE and VPTR)

- If class has at least one virtual function, compiler will add *_vptr to class. Compiler will not insert *_vptr to derived class. Derive class automatically will inherit it from base class.
- Derive class's function will also become virtual if it is overriding base class function.
- Compiler will create Vtable(static array of function pointers) for each class.
- If n number of objects are created, then n number of *_vptr will be added.
- Vptr contains address of vtable, if object is of Base class then vptr will point to Base class's vtable and if
- Object is of derived class then vptr will point to derived class's vtable.

Virtual function working concept

```
class A
{
  * _vptr
  public:
  void f1() {...}
  virtual void f2() {...}
  virtual void f3() {...}
  virtual void f4() {...}
};
class B: public A
{
  public:
  void f1() {...}
  void f2() {...}
  void f4(int x) {...}
};
```

vtable
| f2 | f3 | f4 | |

o1

p → vptr

vtable
| f2 | f3 | f4 | |

```
main()
{
  A *p,o1;
  p=&o1;
  p->f1(); EB
  p->f2(); LB
  p->f3(); LB
  p->f4(); LB
  p->f4(5); EB
}
```

Disadvantages of Virtual functions:

- slower -- The function call takes slightly longer due to the virtual mechanism, and it also makes it more difficult for the compiler to optimize because it doesn't know exactly which function is going to be called at compile time.
- harder to debug -- In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from. Or, to figure out why a function isn't being called if someone overrode it with a new virtual function.

----
local scope, global scope, block scope:

Constructor: is a member function of a class which initializes objects of a class. Constructor is automatically called when object (instance of class) is created. It does not return any value. It uses the same class name. Constructor can be overloaded. Constructor can't be const, static, volatile, and virtual. When parameterized constructor is called, we must define default constructor. When we use new operator, constructor is called.

In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. Compiler should know the class type before object initialization.

Constructor in C++

Default	Parameterized	Copy
Class_name()	Class_name(parameters)	Class_name(const Class_name old_object)

ƏG
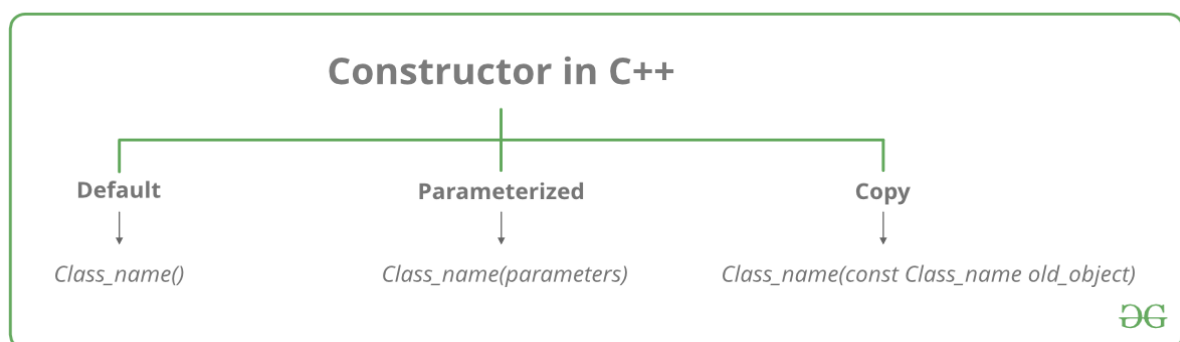
Conversion constructor: if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor. There are constructors that convert types of its parameter into a type of the class. The compiler uses these constructors to perform implicit class-type conversions. These conversions are made by invoking the corresponding constructor with matches the list of values/objects that are assigned to the object.

Usage of conversion constructor:
1) In return value of a function:
When the return type of a function is a class, instead of returning a object, we can return a *braced-init-list*, now since the return type is a class instance, a object of that class is created with the *braced-init-list*, given that the class has a corresponding conversion constructor.

Example:

```
MyClass create_object(int x, int y)
{
  return {x, y};
}
```

2) As a parameter to a function:

When a function's parameter type is of a class, instead of passing a object to the function, we can pass a *braced-init-list* to the function as the actual parameter, given that the class has a corresponding conversion constructor.

An example :

```
void display_object(MyClass obj)
{
obj.display();
}

// This function is invoked in the main function with a braced-init-list with two integers as the parameter.
// e.g. :
// display_object({10, 20});
```

Example:

```
class MyClass {
        int a, b;

public:
        MyClass(int i, int y)
        {
                a = i;
                b = y;
        }
        void display()
        {
                std::cout << " a = " << a << " b = " << b << "\n";
        }
        //function returning class object.
        MyClass create_object(int x, int y)
        {
                return {x, y};
        }
        //function takes class object
        void display_object(MyClass obj)
        {
                obj.display();
        }
};

int main()
{
        MyClass object(10, 20);
        object.display();

        // Multiple parameterized conversion constructor is invoked.
        object = { 30, 40 };
        object.display();
        MyClass ob = object.create_object(50, 60); //will call create_object function, which returns class object.
        ob.display();
        ob.display_object({70, 80}); //This function is invoked in the main function with a braced-init-list with two integers as the
parameter.
        return 0;
}
```

Conversion operator:

```
class MyClass
{
        int a;

public:
        MyClass(int i)
        {
                a = i;


        }
        operator int()
        {
                return a;
        }
```

```
};

int main()
{
        MyClass object(10);
        int i = object;
        cout<<"value= "<<i<<endl;
}
```

Explicit constructor---> if constructor is declared as explicit, only direct object initialization will be allowed.
EX:
class A
{ public:
   explicit A();
   explicit A(int);
   explicit A(const char*, int = 0);
};

   below will be legal to call constructors.
   A a1;
   A a2 = A(1); ---> called copy initialization so need to typecast with A
   A a3(1); --> direct initialization is allowed
   A a4 = A("Venditti");
   A* p = new A(1);
   A a5 = (A)1;
   A a6 = static_cast<A>(1);

----
copy constructor: why we use const class name&---> Do not want copy constructor to be called infinitely. And NULL cannot be assigned to it.
When copy constructor is called:
1. When an object is constructed based on another object of the same class.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object of the class is returned by value.
4. When the compiler generates a temporary object.
It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases

```
#include <iostream>
using namespace std;
class Test
{
        int x;
        public:
        Test(int x):x(x)
        {
                cout<<"conversion constructor is called!"<<endl;
        }
        Test(const Test& t)
        {
                x = t.x;
                cout<<"copy constructor is called!"<<endl;
        }
        Test(const Test&& t)
        {
                cout<<"move constructor is called!"<<endl;
        }

};

void fun1(Test t1)
{

}

Test fun2(Test a1)
{
        return a1;
}

int main()
{
        Test a1(10); //conversion ctor is called.
        //Test a2 = a1; //call copy constructor
        //fun1(a2); //will call copy constructor: "Test t1 = a2"
        Test a3 = fun2(a1); //will call copy constructor for "Test a1 = a1" and move constructor for      return a1(returning
object by value).
        //If we do not provide move constructor then, copy constructor will be called...
```

```
            return 0;
}
```

- **Can we make copy constructor private?**
  Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

//Temoprary object creation: Unnamed object created by compiler. It has impact on performance.
Temporary object is created for below scenarios:

- **Evaluation of expression:**
  Test a1(10); Test a2(20)
  Test a3 = a1 + a2; // must create temporary object
  //compiler will do the following steps:
  //Test _tmp = a1 + a2;
  //a3 = _tmp;

- **Argument passing:**
  int func(const int& x) //function definition
  func(3) // passing r value to function, compiler will create temporary object.
  //compiler will do the following steps:
  // int _tmp = 3
  //func(_tmp)

- **Function returns (return by value):**
  string s = getMessage()
  //compiler will do the following steps:
  //string _tmp(getMessage())
  //string s = _tmp;

- **Reference initialization:**
  int i = 2;
  const double& d = i;
  //compiler will do the following steps:
  //double _tmp = i;
  // const double& d = _tmp;

Copy elision in C++: (Copy omission) is a compiler optimization technique that avoids unnecessary copying of objects. Now a days, almost every compiler uses it.

```
class B
{
public:
        B(const char* str = "\0") //default constructor
        {
                cout << "Constructor called" << endl;
        }

        B(const B &b) //copy constructor
        {
                cout << "Copy constructor called" << endl;
        }
};

int main()
{
        B ob = "copy me";
        return 0;
}
```

Why copy constructor is not called?
According to theory, when the object "ob" is being constructed, one argument constructor is used to convert "copy me" to a temporary object & that temporary object is copied to the object "ob". So the statement
    B ob = "copy me";
should be broken down by the compiler as
    B ob = B("copy me"); => **B ob = B temp = B("copy me");**
However, most of the C++ compilers avoid such overheads of creating a temporary object & then copying it.

The modern compilers break down the statement

    B ob = "copy me"; //copy initialization

as

    B ob("copy me"); //direct initialization

and thus eliding call to copy constructor.

If "-fno-elide-constructors" option is used, first default constructor is called to create a temporary object, then copy constructor is called to copy the temporary object to ob.
Ex:
g++.exe test.cpp -fno-elide-constructors -o test

Finally, this problem was solved by Move semantics.........

lvalue and rvalue:
- -

// lvalues reference:
lvalue reference can be bound to only lvalue, not to rvalue.
int i = 10; --> 10 is rvalue
int &x = i; -> correct assigning lvalue to lvalue reference.
int &x = 10; //wrong can not assign rvalue to lvalue reference
int foo(int&i) { }
foo(10); //wrong can not assign rvalue to lvalue reference
foo(i); -> correct
Can bind rvalue with const lvalue reference.
Const int& x= 10;

//rvalue reference:
rvalue is represented using &&
an expression is rvalue if it results in temporary object.
Ex1:
int i =10;
int &&x = 10;
int &&y = foo(10);
Ex2:
void f(int& l) {cout<<"lvalue is called";}
void f(int&& r) {cout<<"rvalue is called";}
int main()
{
    int i= 10;
    f(i); //will call lvalue function
    f(10); //will call rvalue function
    f(std::move(i)); // will call rvalue function, since move converts lvalue to rvalue. Here i is lvalue converted in rvalue. move(i) will take value of i
that is 10.
}

What is a Move Constructor?

The copy constructor in c++ work with the l-value references and copy semantics(copy semantics means coping the actual data of the object to another object rather than making another object to point the already existing object in the heap). While move constructors work on the r-value references and move semantics (move semantics involves pointing to the already existing object in the memory).

On declaring the new object and assigning it with the r-value, firstly a temporary object is created, and then that temporary object is used to assign the values to the object. Due to this the copy constructor is called several times and increases the overhead and decreases the computational power of the code. To avoid this overhead and make the code more efficient we use move constructors.

Why are Move Constructors used?
Move constructor moves the resources in the heap, i.e., unlike copy constructors which copy the data of the existing object and assigning it to the new object, move constructor just makes the pointer of the declared object to point to the data of temporary object and nulls out the pointer of the temporary objects. Thus, move constructor prevents unnecessarily copying data in the memory.

Work of move constructor looks a bit like default member-wise copy constructor but in this case, it nulls out the pointer of the temporary object preventing more than one object to point to same memory location.

"A move constructor allows the resources owned by an rvalue object to be moved into an lvalue without creating its copy."

Below is the program without declaring the move constructor:

```
class A{
  int *ptr;
public:
  A(int x){
    // Default constructor
    cout << "Calling Default constructor\n";
    ptr = new int ;
              *ptr = x;
  }
```

```cpp
  A( const A & obj){
    // Copy Constructor
    // copy of object is created
    this->ptr = new int;
                *ptr = *(obj.ptr);
    // Deep copying
    cout << "Calling Copy constructor\n";
  }

  ~A(){
    // Destructor
    cout << "Calling Destructor\n";
    delete ptr;
  }

};

int main()
 {

  vector <A> vec;
  vec.push_back(A(10));
  return 0;
}
```

Solution with Move semantics:

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

class A
{
        int *ptr;
        public:
        A(int x)
        {
        // Conversion constructor
        cout << "Calling conversion constructor\n";
        ptr = new int ;
                    *ptr = x;
        }


        A( const A & obj)
        {
        // Copy Constructor
        // copy of object is created
        this->ptr = new int;
                    *ptr = *(obj.ptr);
        // Deep copying
        cout << "Calling Copy constructor\n";
         }

        A ( A && obj)
        {
        // Move constructor
        // It will simply shift the resources,
        // without creating a copy.
         cout << "Calling Move constructor\n";
        this->ptr = obj.ptr;
        obj.ptr = NULL;
        }

        ~A()
        {
        // Destructor
         //cout << "Calling Destructor\n";
         delete ptr;
         }
        void display()
        {
                    cout<<"value is:"<<*ptr<<endl;
        }

};

A fun( A a1)
{
        return a1;
```

```
}
int main()
{
                //A a1(10); //calling conversion constructor
                //A a2 = fun(a1);          // will call copy ctor for A a1 = a1(function parameter) but when a1 is returned again it will call
                //copy constructor if move constructor is not defined... we can fix this prob by defining move constructor. move ctor
                //will move rvalue's memory to lvalue's memory and also will set null for rvalue's pointer.
                //a2.display();
                vector <A> vec;
                vec.push_back(A(10)); // vec.push_back(A _temp = A(10)) : will call conversion operator and then copy constructor to
copy the
                //_temp to function.
   return 0;
}
```

Ex 2:
```
#include <iostream>
#include <cstring>
using namespace std;

class A
{
                int size;
                double* arr;
public:
   A()
   {

   }
   A(const A& rhs)
   {
      size = rhs.size;
      arr = new double[size];
      for(int i =0; i< size; i++)
      {
         arr[i] = rhs.arr[i];
      }
      cout<<"copy ctor called!"<<endl;
   }
   A(A&& rhs)
   {
      size = rhs.size;
      arr = rhs.arr;
                              rhs.arr = nullptr;
      cout<<"move ctor called!"<<endl;
   }

};

void foo(A a)
{

}

void foo_by_ref(A& a)
{

}

A createA()
{

}

int main(int argc, char* argv[])
{
                A a1 = createA();
                foo_by_ref(a1); // will call no constructor....
                foo(a1); // will call costly copy constructor
                foo(createA()); //createA() will return temporary value to function foo.., will call move ctor
                //foo(move(a1)); can move rhs to lhs if object a1 is no more required after this ops...
```

```
        return 0;
}
```

Destructor: It is called when object's scope goes off. When we call delete, destructor is called.

- **When is destructor called?**
  A destructor function is called automatically when the object goes out of scope:
  (1) the function ends
  (2) the program ends
  (3) a block containing local variables ends
  (4) a delete operator is called

- **Can a destructor be virtual?**
  Yes, Infect, it is always a good idea to make destructors virtual in base class when we have a virtual function

Private Destructor:

```
class Test
{
private:
        ~Test() {}
};
int main()
{
        Test t;
}
```

Dynamic memory allocation:

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;        // Pointer initialized with null
pvalue  = new char[20];      // Request memory for the variable
```

To remove the array that we have just created the statement would look like this −

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

```
int*p = new(nothrow) int[5];
        if(p == nullptr)
        {
                cout<<"error"<<endl;
        }
        else
        {
                cout<<"successful"<<endl;
        }
```

Malloc vs New:

new	malloc()
calls constructor	does not call constructors
It is an operator	It is a function
Returns exact data type	Returns void *
on failure, Throws bad_alloc exception	On failure, returns NULL
size is calculated by compiler	size is calculated manually

Delete and Free:
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int x;
        int *ptr1 = &x;
        int *ptr2 = (int *)malloc(sizeof(int));
        int *ptr3 = new int;
        int *ptr4 = NULL;

        /* delete Should NOT be used like below because x is allocated
                on stack frame */
        delete ptr1;

        /* delete Should NOT be used like below because x is allocated
```

```
                using malloc() */
        delete ptr2;

        /* Correct uses of delete */
        delete ptr3;
        delete ptr4;

        getchar();
        return 0;
}
```

Helper function: Private member functions cannot be called using object so those should be called inside public function.

This pointer: is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have **this** pointer, because friends are not members of a class. Only member functions have **this** pointer.

The type of this pointer is either ClassName * or const ClassName *, depending on whether it is inspected inside a non-const or const method of the class ClassName.

Pointer this is not an lvalue.

Reference: Variable is declared as reference, it becomes alternative name for that var.
int x = 10; int &ref = x; (If we change reference var value, original x's value is changed.
Points to take care for reference variable:

- we cannot assign constant value and NULL value to reference.
  Ex: int& val = 10 (wrong)
  int& val = NULL (wrong)

- we should not return local variable as reference variable.
  Ex: int& fun()
      {
          int y = 5;
          return y; →wrong, returning local variable as a reference…
      }

When to use reference:

- Modify the passed parameters in function

- Avoid copy of large structures

- To avoid object slicing

- Range based For loop

Friend function: Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:
a) A method of another class
b) A global function

- Breaks the encapsulation, friend is not mutual with classes, friend function cannot be inherited.

- Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.

- Friendship is not inherited.

    Ex:

```
class A {
int a;

public:
A() { a = 0; }

// global friend function
friend void showA(A&);
};

void showA(A& x)
{
// Since showA() is a friend, it can access
// private members of A. if pass reference of object then can modify it.
std::cout << "A::a=" << x.a;
}

int main()
```

```
{
A a;
showA(a);
return 0;
}
```

Class B is friend of A:
```
class A
{
        int x;
        public:
         A(int x):x(x){}
         //friend void Access(A&);
         friend class B;
         int getvalue()
         {
                 return x;
         }
};

class B
{
        int y;
        public:
         B(int y):y(y){}
         checkvalue(A a1)
         {
                 cout<<a1.x<<endl;
         }

};

int main()
{


        A a1(10);
        B b1(20);
        //Access(a1);
        b1.checkvalue(a1);
        int val = a1.getvalue();
        cout<<"updated value"<<val<<endl;


}
```

Templates in C++

Is used for Generic function like want to write add() for integer/float/double only one method is sufficient.

How templates work?
Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates
and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Two types of templates are used: function templates and class templates

Smart pointers--> We need smart pointers in case of dynamic memory allocation and pointer is pointing to that memory. If exception occurs in between then memory will not be freed. Smart pointer ensures no memory leak.
unique_ptr: Allows only one owner of underlying pointer.
If you are using a unique pointer then if one object is created and pointer P1 is pointing to this one then only one pointer can point this one at one time. So we can't share with another pointer, but we can transfer the control to P2 by removing P1.
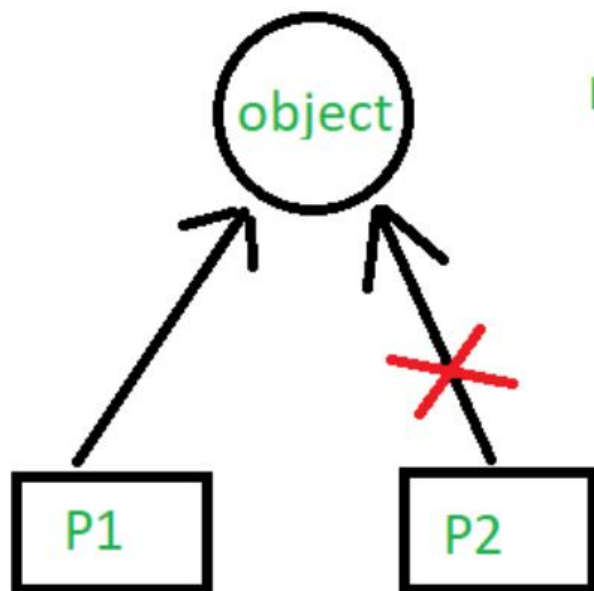
Note: But we can transfer the control to P2 by removing P1

Ex:
#include <iostream>
using namespace std;
#include <memory>

class Rectangle {
int length;

```cpp
int breadth;

public:
Rectangle(int l, int b)
{
length = l;
breadth = b;
}

int area()
{
return length * breadth;
}
};

int main()
{

unique_ptr<Rectangle> P1(new Rectangle(10, 5));
cout << P1->area() << endl; // This'll print 50

unique_ptr<Rectangle> P2;
P2 = move(P1);

// This'll print 50
cout << P2->area() << endl;

// cout<<P1->area()<<endl;
return 0;
}
```
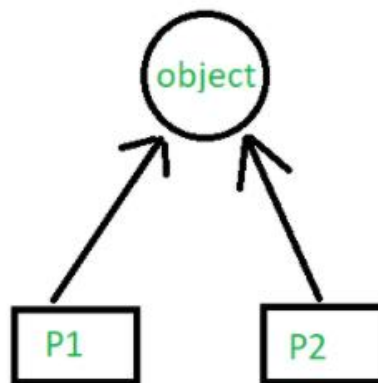
shared_ptr: Allows multiple owner of same pointers
If you are using shared_ptr then more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** and can be checked using **use_count()**

Note:It maintains a Reference Counter by using use_count() method

Here Reference Counter is 2

```cpp
#include <memory>

class Rectangle {
int length;
int breadth;

public:
Rectangle(int l, int b)
{
length = l;
breadth = b;
}

int area()
{
return length * breadth;
}
};

int main()
{
```

```
shared_ptr<Rectangle> P1(new Rectangle(10, 5));
// This'll print 50
cout << P1->area() << endl;

shared_ptr<Rectangle> P2;
P2 = P1;

// This'll print 50
cout << P2->area() << endl;

// This'll now not give an error,
cout << P1->area() << endl;

Ex:

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

class A
{
        int x;
        public:
        A(){}

};

shared_ptr<A> fun()
{
        shared_ptr<A> ptr1(new A);
        shared_ptr<A> ptr2 = ptr1;
        return ptr2;
}

int main()
 {

    shared_ptr<A> ptr = fun();
        shared_ptr<A> ptr3 = ptr;
    cout<<"reference count="<<ptr.use_count()<<endl;
    return 0;
}
```
Weak_Ptr: special type of shared_ptr, which does not count reference.

Default argument--> A default argument is a value provided in a function definition that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.
Ex:
```
// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
cout << sum(10, 15) << endl;
cout << sum(10, 15, 25) << endl;
cout << sum(10, 15, 25, 30) << endl;
return 0;
}
```
//During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.

Static data member and function-->

- Accessible by both non static data member function and static member function.

- It is stored in global place in memory and only single copy is shared b/w all objects.

- It is initialized to zero when the first object of its class is created.

- static member functions do not have **this pointer**. If we want any variable to be shared b/w all objects ex: app_running_status. Even other class can access by creating object and can access the same value.

- A static member function cannot be virtual.

- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.

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

class A
{
public:
        A() { cout << "A's Constructor Called " << endl; }
};

class B
{
        static A a;
public:
        B() { cout << "B's Constructor Called " << endl; }
};

A B::a;
int main()
{
        B b;
        return 0;
}
```
o/p: A's Constructor Called //first global code will be executed, A B::a, so A's constructor will be called first.
B's Constructor Called
----
object slicing--> Slicing" is where you assign an object of a derived class to an instance of a base class, thereby losing part of the information wre you assign an object of a derived class to an instance of a base class, thereby losing part of the information . To avoid object slicing, we must use reference.
```cpp
class Base { int x, y; };

class Derived : public Base { int z, w; };

int main()
{
Derived d;
Base b = d; // Object Slicing, z and w of d are sliced off
}
```

Exception handling:
The try statement allows you to define a block of code to be tested for errors while it is being executed.
The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

```cpp
try {
 int age = 15;
 if (age > 18) {
 cout << "Access granted - you are old enough.";
 } else {
 throw 505;
 }
}
catch (int myNum) {
 cout << "Access denied - You must be at least 18 years old.\n";
 cout << "Error number: " << myNum;
}


catch(...)

{

}
```

```cpp
//Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.
//Whenever exception is cached, after try block all statements will be executed further.
#include <iostream>
using namespace std;
#if 0
int main()
{
int x = -1;
```

```cpp
// Some code
cout << "Before try \n";
try {
        cout << "Inside try \n";
        if (x < 0)
        {
                throw x;
                cout << "After throw (Never executed) \n";
        }
}
catch (int x )
{
        cout << "Exception Caught \n";
}

cout << "After catch (Will be executed) \n";
return 0;
}
#endif
//Output:
```
//Before try
//Inside try
//Exception Caught
//After catch (Will be executed)

//There is a special catch block called 'catch all' catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception,
// but there is no catch block for int, so catch(…) block will be executed

```cpp
/*
int main()
{
        try {
        throw 10;
        }
        catch (char *excp) {
                cout << "Caught " << excp;
        }
        catch (...) {
                cout << "Default Exception\n";
        }
        return 0;
}
*/
```
//If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program,
// a char is thrown, but there is no catch block to catch a char.
```cpp
/*
int main()
{
        try {
        throw 'a';
        }
        catch (int x) {
                cout << "Caught ";
        }
        return 0;
}
*/
```
//terminate called after throwing an instance of 'char'
//This application has requested the Runtime to terminate it in an
//unusual way. Please contact the application's support team for
//more information.

//A derived class exception should be caught before a base class exception

//When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

```cpp
class Test {
public:
Test() { cout << "Constructor of Test " << endl; }
~Test() { cout << "Destructor of Test " << endl; }
};

int main() {
try {
        Test t1;
        throw 10;
```

```
} catch(int i) {
        cout << "Caught " << i << endl;
    }
}
```

o/p:
Constructor of Test
Destructor of Test
Caught

Throw exception from constructor:
Example for constructor throws an exception and catch is present in main.
```cpp
#include <iostream>

class A
{
    private:
        int a;
    public:
        A() { a = 7; throw 42; }
        int getA() { return a; }
};

int main (void) {
    A *ptr;
    try {
        ptr = new A();
    } catch (int b) {
        std::cout << "Exception: " << b << '\n';
        return -1;
    }
    std::cout << "Value: " << ptr->getA() << '\n';
    return 0;
}
```

Example for constructor throws an exception and catch is present in constructor.
```cpp
#include <iostream>

class A {
    private:
        int a;
    public:
        A() {
            try {
                a = 7;
                throw 42;
            } catch (int b) {
                std::cout << "Exception A: " << b << '\n';
                throw;
            }
        }
        int getA() {return a;}
};

int main(void) {
    A *ptr;
    try {
        ptr = new A();
    } catch (int b) {
        std::cout << "Exception B: " << b << '\n';
        return -1;
    }
    std::cout << "Value: " << ptr->getA() << '\n';
    return 0;
}
```

Write your own Exception class to override the exception class functionality:
```cpp
#include <iostream>
#include <exception>

class MyException : public std::exception
{
    public:
    MyException()
    {
        std::cout << "myException constructor is called"<<std::endl;
    }
                const char * what () const throw ()
    {
```

```cpp
            return "C++ Exception";
    }
};

int main()
{
    try
    {
            throw MyException();
    }
    catch (MyException& e)
    {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    }
    catch (std::exception& e)
    {
            // Other errors
    }
}
```

The process of removing function entries from function call stack at run time is called <u>Stack Unwinding</u>. Stack Unwinding is generally related to Exception Handling. In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

```cpp
/* File Handling with C++ using ifstream & ofstream class object*/
/* To write the Content in File*/
/* Then to read the content of file*/
#include <iostream>

/* fstream header file for ifstream, ofstream,
fstream classes */
#include <fstream>

using namespace std;

// Driver Code
int main()
{
        // Creation of ofstream class object
        ofstream fout;

        string line;

        // by default ios::out mode, automatically deletes
        // the content of file. To append the content, open in ios:app
        // fout.open("sample.txt", ios::app)
        fout.open("sample.txt");

        // Execute a loop If file successfully opened
        while (fout) {

                // Read a Line from standard input
                getline(cin, line);

                // Press -1 to exit
                if (line == "-1")
                        break;

                // Write line in file
                fout << line << endl;
        }

        // Close the File
        fout.close();

        // Creation of ifstream class object to read the file
        ifstream fin;

        // by default open mode = ios::in mode
        fin.open("sample.txt");

        // Execute a loop until EOF (End of File)
        while (fin) {

                // Read a Line from File
                getline(fin, line);
```

```
                // Print line in Console
                cout << line << endl;
        }

        // Close the file
        fin.close();

        return 0;
}
```

Type Casting:
Static Cast: This is the simplest type of cast which can be used. It is a **compile time cast**. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions (or implicit ones).
For e.g.

```
int main()
{
float f = 3.5;
int a = f; // this is how you do in C
int b = static_cast<int>(f);
cout << b;
}


int main()
{
int a = 10;
char c = 'a';
// pass at compile time, may fail at run time
int* q = (int*)&c;
int* p = static_cast<int*>(&c);
return 0;
}
```

//Use static_cast when conversion b/w types is through conversion operator and conversion constructor.
```
class Int {
    int x;

public:
    Int(int x_in = 0)
        : x{ x_in }
    {
        cout << "Conversion Ctor called" << endl;
    }
    operator string()
    {
        cout << "Conversion Operator" << endl;
        return to_string(x);
    }
};
int main()
{
    Int obj(3);
    string str = obj; will call conversion operator
    obj = 20; will call conversion constructor
    string str2 = static_cast<string>(obj);
    obj = static_cast<Int>(30);
    return 0;
}
```
o/p:
Conversion Ctor called

Conversion Operator

Conversion Ctor called

Conversion Operator

Conversion Ctor called

//Static_cast avoids cast from derived to private Base pointer:

Class Base{};

Class Derived:private Base{}; privately inherited…

Int main()

{

       Derived d1;

       Base *p1 = (Base*)&d1; //Allowed at compile time: c style conversion

       Base *p1 = static_cast<Base*>(&d1) not allowed, base is inaccessible base of derived.

}

//**Use for all upcasts but not for confused downcast.**

```
class Base {};
class Derived1: public Base {};
class Derived2: public Base {};
int main()
{
          Derived d1, d2;
          Base *b1 = static_cast<Base*>(&d1); ->Upcast: valid
          Base *b2 = static_cast<Base*>(&d2); ->Upcast : valid
          Derived *d1p = static_cast<Derived1*>(&b2); -> Downcast: will compile but might create issue at run time.
          Derived *d2p = static_cast<Derived2*>(&b1); ); -> Downcast: will compile but might create issue at run time.


}
```

Dynamic_cast: can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class. Class should be polymorphic:should have at least one virtual function.

```
Ex:
 class CBase { };
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base(upcasting)
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived(downcasting). Possible if Baseclass* points to derived class object.
```

const_cast: This type of casting manipulates the constness of an object, either to be set or to be removed.

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

void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```
reinterpret_cast: converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

```
Ex:  class A {};

class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

The mutable storage class specifier in C++ (or use of mutable keyword in C++):

C++ also supports auto, register, static, extern storage class specifiers. In addition to this C++, adds one important **storage class specifier whose name is mutable**.

What is the need of mutable?

1) Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don't want the function to update other members of class / struct. The keyword mutable is mainly used to allow a particular data member of const object to be modified. When we declare a function as const, the this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.

Example1:

```
class Customer
{
char name[25];
mutable char placedorder[50];
int tableno;
mutable int bill;
public:
Customer(char* s, char* m, int a, int p)
{
strcpy(name, s);

strcpy(placedorder, m);
tableno = a;
bill = p;
}
void changePlacedOrder(char* p) const
{
strcpy(placedorder, p);
}
void changeBill(int s) const
{
    bill = s;
}
};
```

Example2:
```
class Test {
public:
int x;
mutable int y;
Test() { x = 4; y = 10; }
};
int main()
{
const Test t1; ☜☜object is constant, we can modify y(mutable) but not x.
t1.y = 20;
cout << t1.y;
return 0;
}
```

String stream in C++ and its applications:

A string stream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin)

string stream class is extremely useful in parsing input.

Example1: // CPP program to count words in a string

```
#include <bits/stdc++.h>
using namespace std;

int countWords(string str)
{
// breaking input into word using string stream
stringstream s(str); // Used for breaking words
string word; // to store individual words

int count = 0;
while (s >> word)
count++;
return count;
}

// Driver code
int main()
```

```cpp
{
string s = "geeks for geeks geeks "
contribution placements";
cout << " Number of words are: " << countWords(s);
return 0;
}
```

Example2: // to count frequencies of words.

```cpp
#include <bits/stdc++.h>
using namespace std;

void printFrequency(string st)
{
// each word it mapped to it's frequency
map<string, int> FW;
stringstream ss(st); // Used for breaking words
string Word; // To store individual words

while (ss >> Word)
FW[Word]++;

map<string, int>::iterator m;
for (m = FW.begin(); m != FW.end(); m++)
cout << m->first << " -> "
<< m->second << "\n";
}
```

Examples: findMinimumCurrency

```cpp
#include <map>
#include <bits/stdc++.h>
using namespace std;

void findMinimumCurrency(int amount, map<int,int>mp)
{
        map<int,int>currency;
        int key_val;
        int j = 0;
        for(auto it = mp.rbegin(); it != mp.rend(); it++)
        {
                if(amount >= it->first)
                {
                        key_val = amount/(it->first);
                        currency[it->first] = key_val;
                        amount = amount - (key_val*(it->first));
                    cout<<"currency[j]"<<currency[it->first]<<endl;
                }
        }
        for(auto it = currency.begin(); it != currency.end(); it++)
        {
                cout<<it->first<<"->"<<it->second<<endl;
        }
}

void printFrequency(vector<int> vec)
{
        int sum = 0;
        map<int, int>mp;
        for(auto it =vec.begin(); it!= vec.end(); it++)
        {
                cout<<*it<<endl;
                mp[*it]++;
        }
        for(auto it =mp.begin(); it!= mp.end(); it++)
        {
                cout<<it->first<<"->"<<it->second<<endl;
                sum = sum + it->first;
        }
        cout<<"Sum"<<sum<<endl;
}
void printFrequency(string& str)
{
        stringstream st(str);
        string word;
        map<string, int>mp;
        while(st>>word)
        {
                mp[word]++;
        }
        for(auto it= mp.begin(); it !=mp.end(); it++)
```

```cpp
                {
                        cout<<it->first<<"->"<<it->second<<endl;
                }
}

int main()
{
        vector<int>vec{1,2,1,3,4};
        map<int,int>mp{{1000,2},{500,2},{200,2},{100,2},{50,2},{10,2},{5,2},{2,2},{1,2}};
        int amount = 878;
        //printFrequency(vec);
        string str = "This is my atta pattu pattu";
        //printFrequency(str);
        findMinimumCurrency(878,mp);
        return 0;
}
```

//Remove duplicate characters from string:
void removeDuplicate(string str)

```cpp
{
   std::vector<char> v(str.begin(), str.end());
   auto end = v.end();
   for (auto it = v.begin(); it != end; ++it)
   {
      end = std::remove(it + 1, end, *it);


   }

   v.erase(end, v.end());
   for (auto it = v.cbegin(); it != v.cend(); ++it)
      std::cout << *it << ' ';
}
```

----

c++11/14 features:
//**1) Auto:** Automatic Type Deduction
//Before C++11, the auto keyword was used for storage duration specification. auto is now a sort of placeholder for a type, telling the compiler it has to deduce the actual type of a variable that is being declared from its initializer. auto can be used for iterators also.
auto i = 42;   // i is an int
auto l = 42LL;   // l is an long long
auto p = new foo(); // p is a foo*

std::map<std::string, std::vector<int>> map;
for(auto it = begin(map); it != end(map); ++it) -->auto it will automatically will take type
{
}

Function return type deduction is implemented in C++14:
auto Correct(int i)
```cpp
{
   if (i == 1)
      return i;        // return type deduced as int


   return Correct(i-1)+i; // ok to call it now
}
```

****
//**2) nullptr:** helps to avoid mistakes which might occur when a null pointer gets interpreted as an integral value. C++ does not allow to implicitly convert void * to other types. But if the compiler tries to
define NULL as ((void*)0), then in the following code:
char *ch = NULL;
****
//**3) // C++ 11 initializer list:**
// vector<int> vec = {10,20,20}; //calling initializer list constructor. Need not to store data using vec.push_back().
//we can create our own initializer list constructor
class myclass
//{
        vector<int>m_vec;
        public:
                myclass(const initializer_list<int>&v)
                {
                        for(initializer_list<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr)
                        {

```
                                    m_vec.push_back(*itr);
                        }
            }

};
int main()
{
            myclass v = {1,2,3,4};
}
```

Use of initializer list: It is used when:

1) To initialize reference variable

2) To initialize const variable

3) When object of one class is created in other class (required, otherwise default constructor of that class is called and if that is not implemented then there will be an error).
   Ex:
   ```
   class A {
               int i,j;
   public:
               A(int, int );
   };

   A::A(int arg1, int arg2) {
               i = arg1;
               j = arg2;
               cout << "A's Constructor called: Value of i: " << i << endl;
   }

   // Class B contains object of A
   class B {
               A a;
   public:
               B(int, int );
   };

   B::B(int x, int y): a(x, y){ //Initializer list must be used, otherwise it will try to call default constructor of class A.
               cout << "B's Constructor called";
   }

   int main() {
               B obj(10,20);
               return 0;
   }
   ```

4) To initialize the members of Base class.
   Ex:
   ```
   #include <iostream>
   using namespace std;

   class A {
               int i;
   public:
               A(int );
   };

   A::A(int arg) {
               i = arg;
               cout << "A's Constructor called: Value of i: " << i << endl;
   }

   // Class B is derived from A
   class B: A {
   public:
               B(int );
   };

   B::B(int x):A(x) { //Initializer list must be used
               cout << "B's Constructor called";
   }

   int main() {
               B obj(10);
               return 0;
   }
   ```

5) To initialize base class data members if function parameter and data member are using same name.

Initialization of data members: In C++, class variables are initialized in the same order as they appear in the class declaration.

****

//4) Range-based for loop: C++11 bring in a new kind of for loop that iterates over all elements of a given range/set of arrays or collection

```
for (declaration : coll/array_name )
{
// statement(s) block;
}
Ex: 1
for ( int i : { 2, 4, 6, 8, 10, 12, 14, 16 } )
{
std:: cout << "The Value :" << i << std:: endl;
}
Ex: 2
std:: vector<int> vect = {10,2,3,4};
for(auto & element :vect)
{
    std::cout<<"element"<<element<<std::endl;
}
```

****

//5) constexpr
```
//By specifying constexpr, we suggest compiler
// to evaluate value at compile time
constexpr int product(int x, int y)
{
    return (x * y);
}

int main()
{
    const int x = product(10, 20);
    cout << x;
    return 0;
}
```

constexpr in c++ 14:
```
//C++11-arrays:
std::array<int, 3> arr = {2, 3, 5};
```

****

//6) C++11 - functors & lambda functions

Functors: Calling object using paranthesis like function call. One advantage of functors over function pointers is that they can hold state. Since this state is held by the instance of the object it can be thread safe (unlike static variables inside functions used with function pointers). The state of a functor can be initialized at construction.

```
**************************functors_example********
#include<iostream>
#include <vector>
#include<algorithm>
using namespace std;

//Implemented functor using struct multiply
/*
struct multiply
{
        private:
        int factor;
        public:
        multiply(){}
        multiply(int x):factor(x){}
        void operator()(int y)
        {
                cout<<"Multiplied vector values:"<<factor*y<<endl;
        }


};
*/
//Implemented functor using class multiply
/*
class multiply
{
        private:
                int factor;
        public:
        multiply(){}
```

```
                multiply(int x):factor(x){}
                void operator()(int y)
                {
                        cout<<"Multiplied vector values:"<<factor*y<<endl;
                }

};
*/

int main()
{
        vector<int>vec = {1,2,3,4};
        int factor = 2;
        //for_each(vec.begin(), vec.end(), multiply(2)); //calling functor(multiply(x)--> value will be passed to constructor)
        //using lambda function
        for_each(vec.begin(), vec.end(), [factor](int y){ //[] will take local variable and () will take vector values
        cout<<"Multiplied vector values:"<<factor*y<<endl;});
        return 0;
}
```

//A 'Lambda' can be defined as a definition of functionality which can be definite inside statements and expressions. So, programmers can use a lambda as an inline function. The minimal lambda function contains no parameters.

```
int main()
 {
    auto sum = [](int x, int y) { return x + y; };
    cout << sum(5, 2) << endl;
    cout << sum(10, 5) << endl;
 }
```

C++ 14 provided generic lambda, suppose you want to have sum function for integer/float etc.
[](auto a, auto b) { return a + b; }

```
int main()
{

   // Declare a generalized lambda and store it in sum
   auto sum = [](auto a, auto b) {
      return a + b;
   };

   // Find sum of two integers
   cout << sum(1, 6) << endl;

   // Find sum of two floating numbers
   cout << sum(1.0, 5.6) << endl;

   // Find sum of two strings
   cout << sum(string("Geeks"), string("ForGeeks")) << endl;

   return 0;
}
```

****
//7) Deleted and Defaulted Functions:
struct A

{

 A()=default; //C++11

 virtual ~A()=default; //C++11

};

is called a defaulted function. The =default; part instructs the compiler to generate the default implementation for the function. Defaulted functions have two advantages: They are more efficient than manual implementations, and they rid the programmer from the chore of defining those functions manually.

The opposite of a defaulted function is a deleted function:

int func()=delete;

Deleted functions are useful for preventing object copying, among the rest. Recall that C++ automatically declares a copy constructor and an assignment operator for classes. To disable copying, declare these two special member functions =delete:

//Delegating Constructors

In C++11 a constructor may call another constructor of the same class:

//8) Override and Final
```
class B

{

public:

   virtual void f(int) const {std::cout << "B::f " << std::endl;}

};


class D : public B

{

public:

   virtual void f(int) {std::cout << "D::f" << std::endl;}

};
int main()
{
        B *b = new D();
        b->f(1); --> this will call base class f() not derived class f(). Since derive class function is not overriding the base class function.
Both the functions are different in signature.
}
```

//if we want to avoid such situation, we will declare derive class function as override, now compiler will give you an error here.
Example with override implementation:
```
class B

{

public:

   virtual void f(int) const {std::cout << "B::f " << std::endl;}

};


class D : public B

{

public:

   virtual void f(int) override {std::cout << "D::f" << std::endl;}

};
```

if you intend to make a method impossible to override any more (down the hierarchy), mark it as final. That can be in the base class, or any derived class. If it's in a derived class, you can use both the override and final specifiers.

//9) static_assert and Type Traits

static_assert performs an assertion check at compile-time. If the assertion is true, nothing happens. If the assertion is false, the compiler displays the specified error message.

```
template <typename T, size_t Size>

class Vector

{

   static_assert(Size < 3, "Size is too small");

   T _points[Size];

};
```

```cpp
int main()
{
  Vector<int, 16> a1;

  Vector<double, 2> a2;

  return 0;

}
```

----

10) noexcept: By declaring a function, a method, or a lambda-function as `noexcept`, you specify that these does not throw an exception and if they throw, you do not care and let the program just crash

Ex:

```cpp
void func1() noexcept;        // does not throw
void func2() noexcept(true);  // does not throw
void func3() throw();         // does not throw
```

■■

```cpp
: that take a variable number of arguments. Variadic function templates are functions which can take multiple number of arguments.
void print()
{
   cout << "I am empty function and "
        "I am called at last.\n" ;
}

// Variadic function Template that takes
// variable number of arguments and prints
// all of them.
template <typename T, typename... Types>
void print(T var1, Types... var2)
{
   cout << var1 << endl ;

   print(var2...) ;
}

// Driver code
int main()
{
   print(1, 2, 3.14, "Pass me any "
        "number of arguments",
           "I will print\n");

   return 0;
}
```

----

placement new: Placement new is a variation <u>new </u>operator in C++. Normal new operator does two things : (1) Allocates memory (2) Constructs an object in allocated memory.

Placement new allows us to separate above two things. In placement new, we can pass a preallocated memory and construct an object in the passed memory.

---std::nothrow:
This constant value is used as an argument for <u>operator new</u> and <u>operator new[]</u> to indicate that these functions shall not throw an exception on failure, but return a *null pointer* instead.

```cpp
// nothrow example
#include <iostream>     // std::cout
#include <new>          // std::nothrow

int main () {
  std::cout << "Attempting to allocate 1 MiB... ";
  char* p = new (std::nothrow) char [1048576];

  if (!p) {             // null pointers are implicitly converted to false
```

```
      std::cout << "Failed!\n";
  }
  else {
      std::cout << "Succeeded!\n";
      delete[] p;
  }

  return 0;
}
```

Composition: Type of association.

Dependent.

One object will carry another object as a value.

One object destroy, other will also destroy.

Strong relationship.

Example: Car object has Engine object. If Car object is destroyed, Engine object will be destroyed.

#include <iostream>

#include <string>

using namespace std;

class Engine

{

       int power;

       public:

       Engine(int power):power(power)

       {

              cout<<"Engine object is created"<<endl;

       }

       ~Engine()

       {

              cout<<"Engine object is destroyed"<<endl;

       }

};

class Car

{

       int model;

       string name;

       Engine eng;

       public:

       Car(string name, int model, Engine eng): name(name), model(model), eng(eng)

       {

              cout<<"Car object is created"<<endl;

       }

       ~Car()

       {

              cout<<"Car object is destroyed"<<endl;

       }

```cpp
};
int main()
{
        Engine e(10);

        Car *c = new Car ("BMW", 134, e);

        delete c;

        return 0;
}
```

Aggregation: Type of association.

Independent.

One object will carry reference of another object.

One object destroy, other will not destroy.

weak relationship.

Example: Person object has reference of Car object. If person object is destroyed, Car object will not be destroyed.

```cpp
class Car
{       int model;
        string name;
        public:
        Car(string name, int model): name(name), model(model){}
        void printCarInfo()
        {
                cout<<model<<name<<endl;
        }
};
class Person
{
        string name;
        Car *mycar;
        public:
        Person() = default;
        Person(string name, Car* mycar):name(name), mycar(mycar)
        {
        }
};
int main()
{
        Car c ("BMW", 134);

        Person *p = new Person("Ankit", &c);

        delete p;

        c.printCarInfo();

        return 0;
```

```
}
```

Implementation:
```
*******************Stack_impl_using_array*************************
#include <iostream>
using namespace std;
#define MAX 10

class stack
{
          int arr[MAX];
          int top;
          public:
                    stack();
                    void push(int item);
                    void pop();
                    void show();
};

stack::stack():top(-1)
{

}

void stack::push(int item)
{
          if(top == MAX-1)
          {
                    cout<<"stack is full"<<endl;
          }
          else
          {
                    arr[++top] = item;
          }
}
void stack::pop()
{
          top--;
}

void stack::show()
{
          for(int i=0; i<top; i++)
          {
                    cout<<arr[i]<<endl;
          }
}

int main()
{
          stack s1;
          s1.push(10);
          s1.push(20);
          s1.push(30);
          s1.show();
          s1.pop();
          s1.show();
          return 0;
}
```

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

class stack
{
          private:
                    int *items;
                    int top;


          public:
                    stack();
                    ~stack();
```

```cpp
                    void push(int data);
                    void pop();
                    void show();
};

stack::stack():items(NULL), top(0)
{


}
stack::~stack()
{
        //free run time allocated memory.
}
void stack::push(int data)
{
        if(NULL == items)
        {
                items = new int[1];
        }
        else
        {
                int *newarray = new int[top+1];
                for(int i=0; i<top; i++)
                {
                        newarray[i] = items[i];
                }
                delete items;
                items = newarray;
        }
        items[top] = data;
        top++;
}

void stack::pop()
{
        top--;
}

void stack::show()
{
        for(int i=0; i<top; i++)
        {
                cout<<"value in stack:"<<items[i]<<endl;
        }
}

int main()
{
        stack s1;
        s1.push(10);
        s1.push(20);
        s1.push(30);
        s1.show();
        s1.pop();
        s1.show();
        return 0;
}

*******************************C++_string_impl*********************
#include <iostream>
#include <cstring>
using namespace std;

class myString
{
        char* str;
        public:
                myString():str(nullptr){}
                ~myString(){}
                myString(char* str1)
                {
                        str = new char[strlen(str1)];
                        strcpy(str, str1);
                }
                myString(const myString& mstr1)
                {
                        str = new char[strlen(mstr1.str)];
                        strcpy(str, mstr1.str);
```

```cpp
			}
			myString& operator=(const myString& mstr1)
			{
					str = new char[strlen(mstr1.str)];
					strcpy(str, mstr1.str);
					return *this;
			}
			myString operator+(const myString& str1)
			{
					myString temp;
					temp.str = new char[strlen(str1.str)+ strlen(str)];
					temp = strcat(str, str1.str);
					return temp;
			}
			void display()
			{
					std::cout<<"current object data"<<str<<std::endl;
			}

};


int main(int argc, char* argv[])
{
		std::cout<<"String is called!"<<std::endl;
		myString s1("Ankit");
		s1.display();
		myString s2 = s1;
		s2.display();
		myString s3("Shukla");
		s3.display();
		myString s4 = s1+s3;


		return 0;
}
```

*****************vector_stl_impl***

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

template <class T>
class myVector
{
		T *items;
		int top;
		public:
		myVector():items(NULL), top(0)
		{

		}
		~myVector()
		{

		}
		myVector(const myVector& vec)
		{
				items = new T[vec.top];
				for(int i=0; i<vec.top; i++)
				{
						items[i] = vec.items[i];
				}
				top = vec.top;
		}
		myVector& operator=(const myVector& vec)
		{
				items = new T[vec.top];
				for(int i=0; i<vec.top; i++)
				{
						items[i] = vec.items[i];
				}
				top = vec.top;
```

```cpp
                    return *this;
            }
            void push_back(int data)
            {
                    //check if stack is empty.
                    if(NULL == items)
                    {
                            items = new T[1];
                    }
                    else
                    {
                            T *newArr;
                            newArr = new T[top+1];
                            for(int i=0; i<top; i++)
                            {
                                    newArr[i]=items[i];
                            }
                            delete items;
                            items = newArr;
                    }
                    items[top]= data;
                    top++;
            }
            void show_data()
            {
                    for(int j=0; j<top; j++)
                    {
                            cout<<"data values:"<<items[j]<<endl;

                    }
            }
            void pop_back()
            {
                    if(NULL!= items)
                    {
                            top--;
                    }
            }

};


int main(int argc, char* argv[])
{

        std::cout<<"Vector is called!"<<std::endl;

        myVector<int>vec1;
        vec1.push_back(10);
        vec1.show_data();
        myVector<int>vec2 = vec1;
        std::cout<<"After copy constructor is called!!"<<std::endl;


        return 0;
}
```