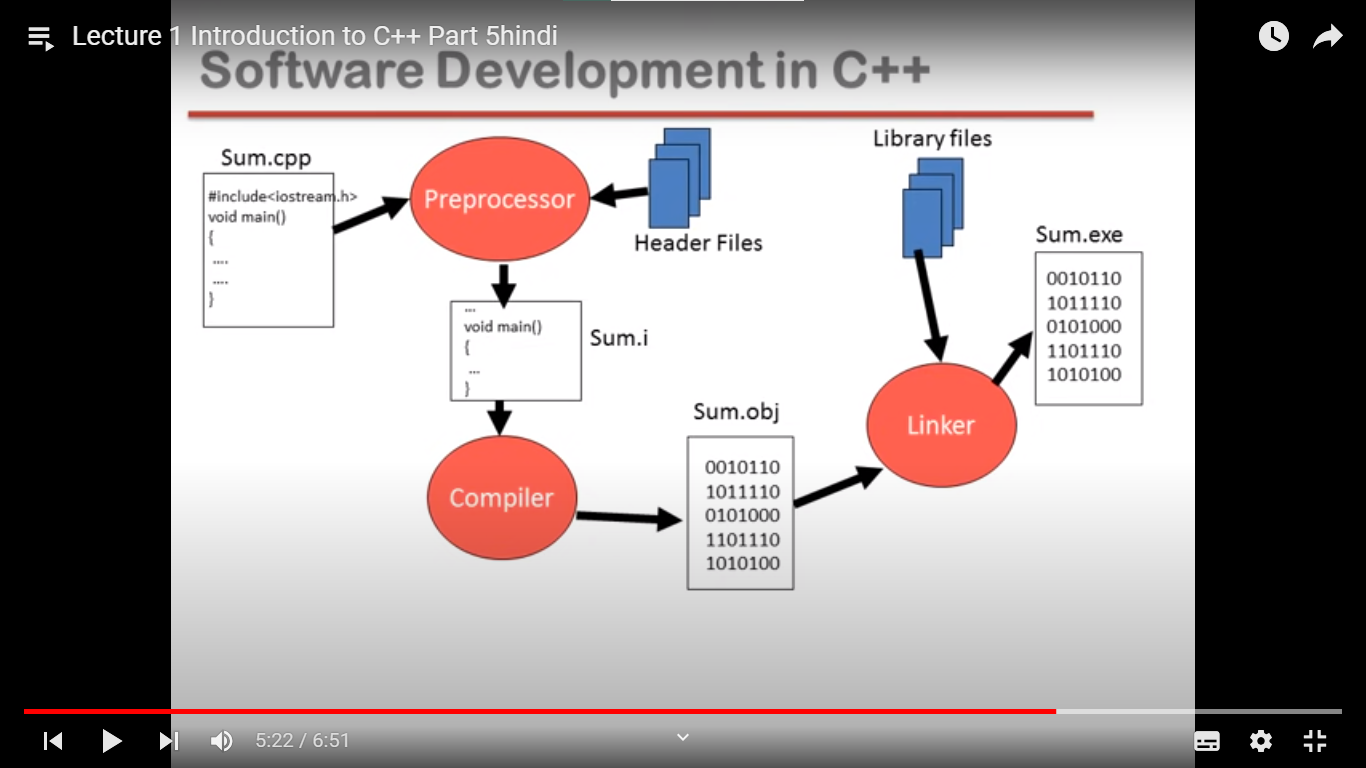
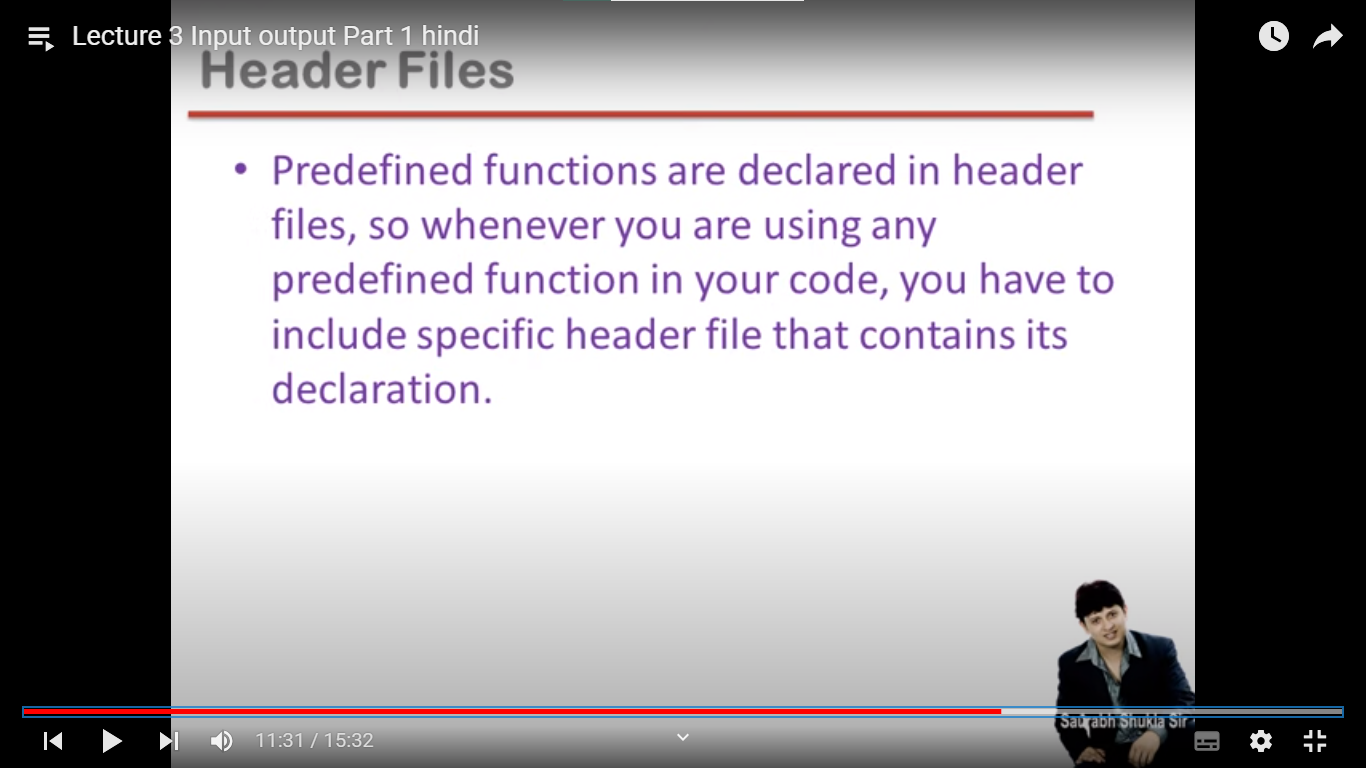
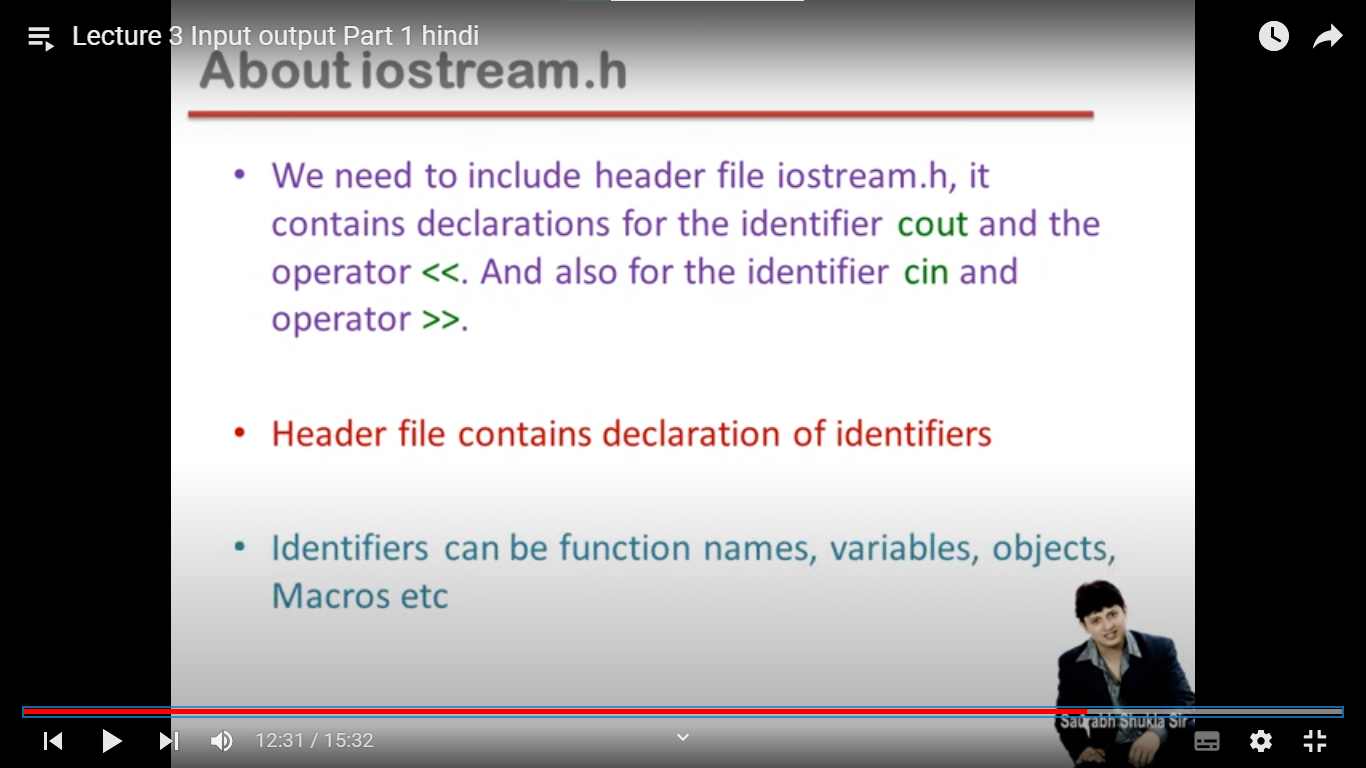
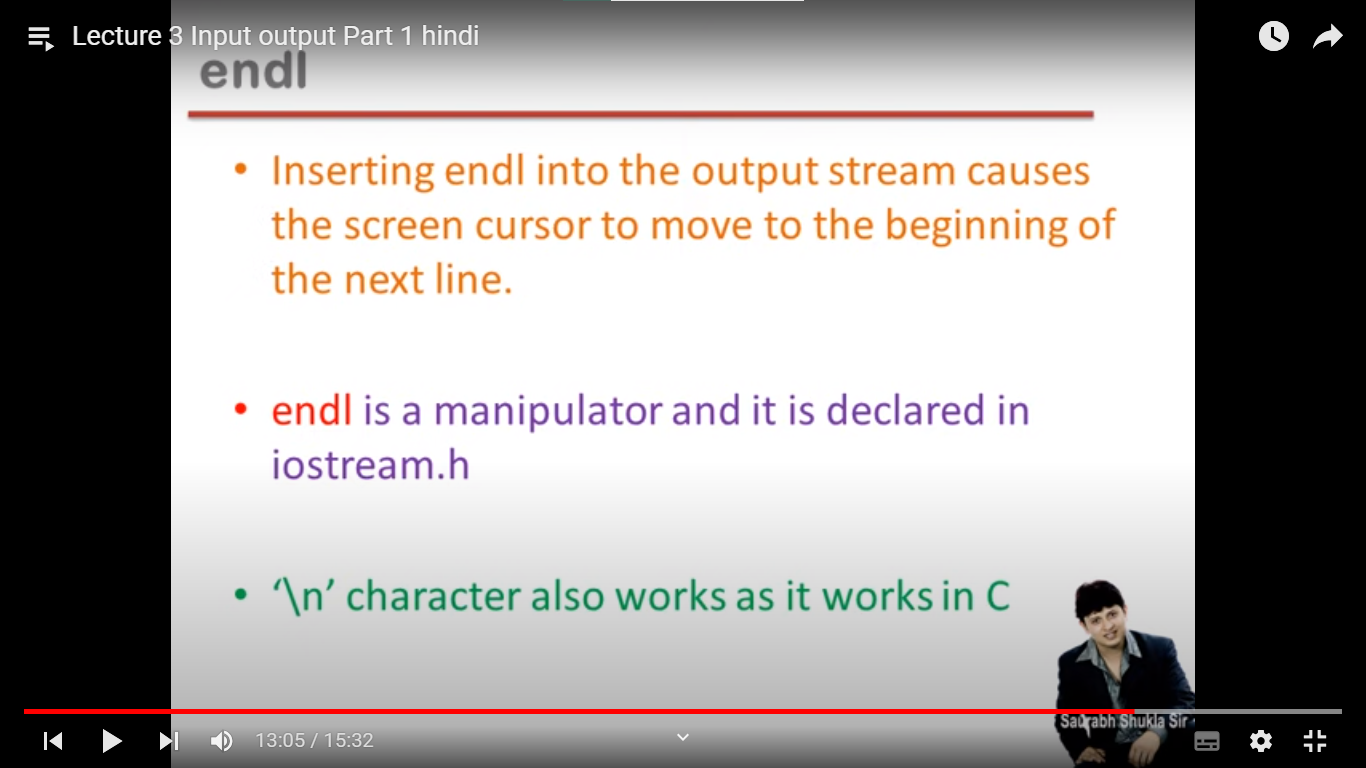
\*C++ Inventor -> Bjarne Stoustrup









\*Difference between C and C++

-->C++ supports both procedural and object-oriented programming, while C supports only procedural-oriented programming. Hence, C is a subset of C++.

-->C++ supports polymorphism, encapsulation, inheritance, overloading, namespace, reference variable, templates, and exception handling, which C does not have.

-->C++ supports STL but C does not. This is the most important.

-> C++ is a superset of C.

-> C follows top down approach of programming.

i.e. we first implement main function and decide the procedure which function has to called when after that we define and implement that functionality.

-> C++ follows bottom up approach of programming.

i.e. we first implement all the functions we need and then at the end we implement main function and procedure of which function to call when.

\*Why C and C++ is platform dependent?

->These languages are platform dependent as after compilation the .exe file which is generated dependent on OS. One file cannot be run on different OS.

->In case of java the java compiler generates .class file which is interpreted by a jvm so it run on any device and hence platform independent.

\*Difference between C++ and Java

->C++ is platform-dependent while Java is platform-independent.

->*C++ supports pointers while Java does not.*

->C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.

->Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.

->C++ supports both call by value and call by reference while Java supports call by value only. There is no call by reference in java.

Java interpreter is a computer program (system software) that implements the JVM. It is responsible for reading and executing the program.

\*Difference between C++ and Python

->C++ tends to have long lines of code while Python has fewer lines of code.

->C++ is faster once compiled as compared to python.

->In C++, the scope of variables is limited within the loops while In Python, variables are accessible even outside the loop.

->when any program starts, it first loads in RAM, in 16 bit architecture our program gets 64kb memory. In memory, we have two sections one is to store

instructions and one is to store data.

/\*

--> Variables are the names of memory locations where we store data.

1. Ordinary Variable:

--> A block gets created in the memory with name x, value 5 and some address(starting at base address something like 201)

int x = 5;

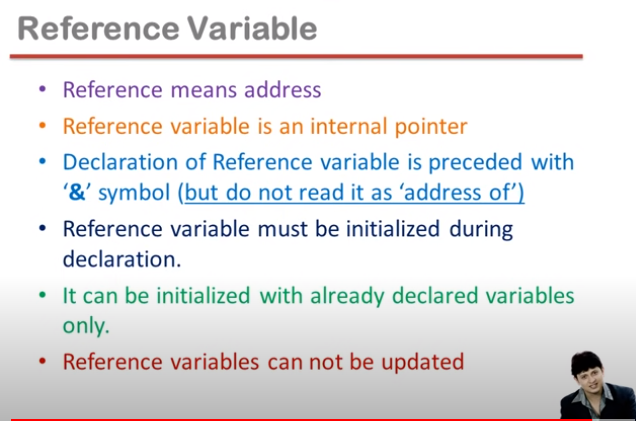
2. Pointer Variable:

--> Used to hold address of some other variable

int \*p;

p = &x; (p = 201 i.e address of x gets assigned to p)

3. Reference Variable: (New in C++, was not in C)



--> It is a new name to an already existing variable

--> To be initialized at declaration only (with some previously created variable and not constant)

--> Contains reference of another variable

--> Reference variables cannot be updated

--> We cannot read it as a address of variable. if & is present on right side then we read it address of.

--> Modern style of pointer. It is an internal pointer

int &y = x;

y++;

cout<<y<<endl; (6 will be printed . it will increment the value of x i.e y is representing x)

couy<<x<<endl;(6 will be printed, it also increments the value of x)

IMP: Address will be used in context of pointers and reference will be used in case of reference variables

cout << "Ordinary Variable:\n";

int x = 3; // Ordinary Variable

cout << "x: " << x << endl;

cout << "\nPointer Variable:\n";

int \*p = &x; // Pointer Variable // To be read as: p holds address(&) of x

cout << "p (Address of x): " << p << endl;

cout << "\*p (Value at address of x): " << \*p << endl;

cout << "\nReference Variable:\n";

int &y = x; // Reference Variable --> Reference of x is stored in y

cout << "y (Value of x): " << y << endl;

cout << "&y (Reference of x): " << &y << endl;

y++; // This increments the value of x

cout << "\nAfter doing y++:\n";

cout << "x: " << x << endl; // Value of x changes to 4

cout << "y (Value of x): " << y << endl; // Value of y changes to 4

cout << "&y (Reference of x): " << &y << endl;

\*/

for pointers concept-> can watch Love babbar videos

https://www.codingninjas.com/studio/guided-paths/pointers

\*Functions: (Memory ki bachat par time jyada leta hein)

-->Function is block of code performing a unit of task.

-->Function has a name, return type and arguments.

-->Function is a way to achieve modularization.

-->Functions are predefined and user defined.

-->Predefined functions are declared in header files and defined in library files.

/\*

What is a function?

--> Block of code performing a task (Helps in achieving modularization)

--> Syntax:

returnType name(arguements){

}

--> Types:

1. Pre-defined(Declared in header files and defined in library files)

2. User-defined

--> Benefits:

1. Easy to Read & Modify

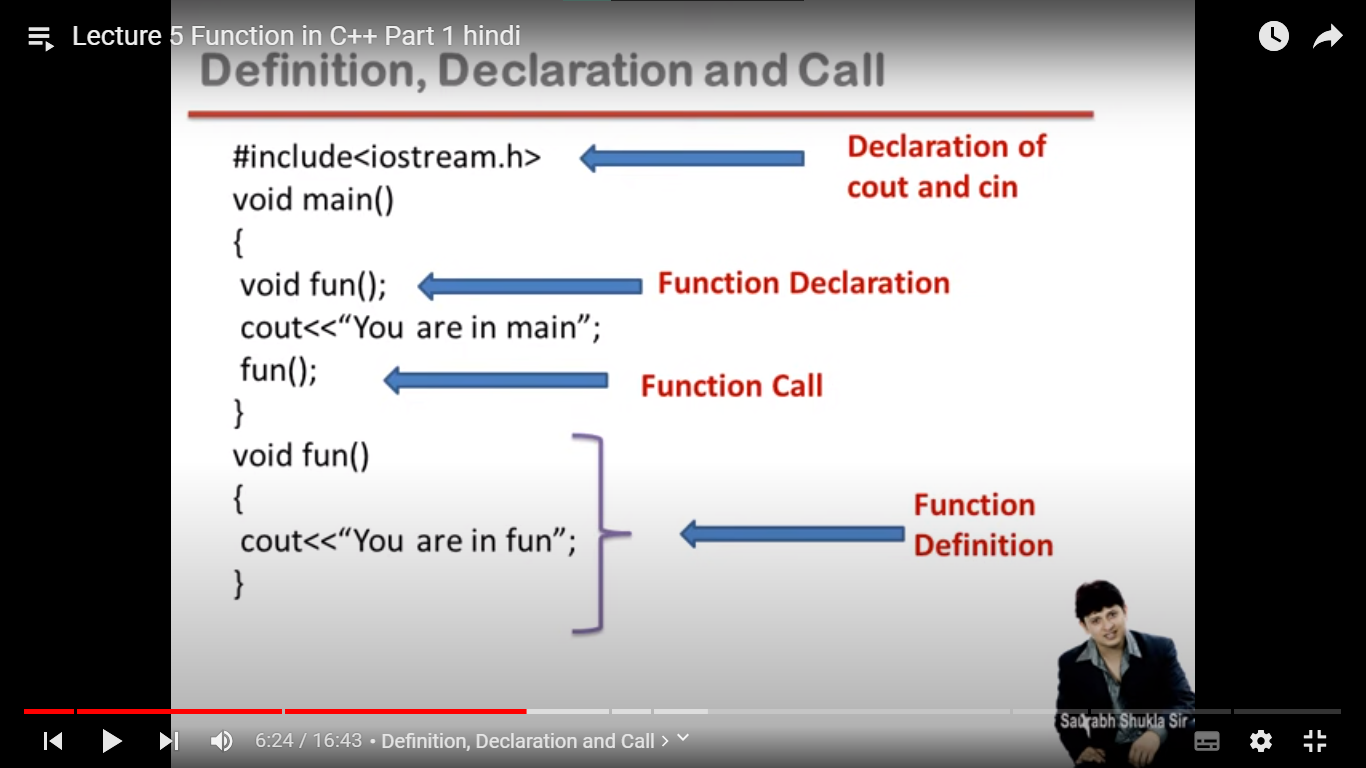
2. Avoid duplication of code

3. Easy to debug codes

4. Better memory utilization (Only the functions in use

are present in RAM)

\*/



e.g.

int sum(int x, int y){-> This x and y are formal arguments

return x+y;

}

int main(){

cout<<sum(a,b)<<endl;-> This a and b are actual arguments

}

/\*

// Function Declaration

void globalFunction(); // can be called anywhere in the program

int main() {

void localFunction(); // Can be only called in main method

cout<<"Calling localFunction() from main: ";

localFunction(); // Function call

globalFunction();

}

// Function Definitions

void globalFunction() {

cout << "In Global Function\n";

/\*

Un-commenting the line below gives the following error

‘localFunction’ was not declared in this scope; did you mean ‘globalFunction’?

\*/

// localFunction();

}

void localFunction() {

cout << "In Local Function\n";

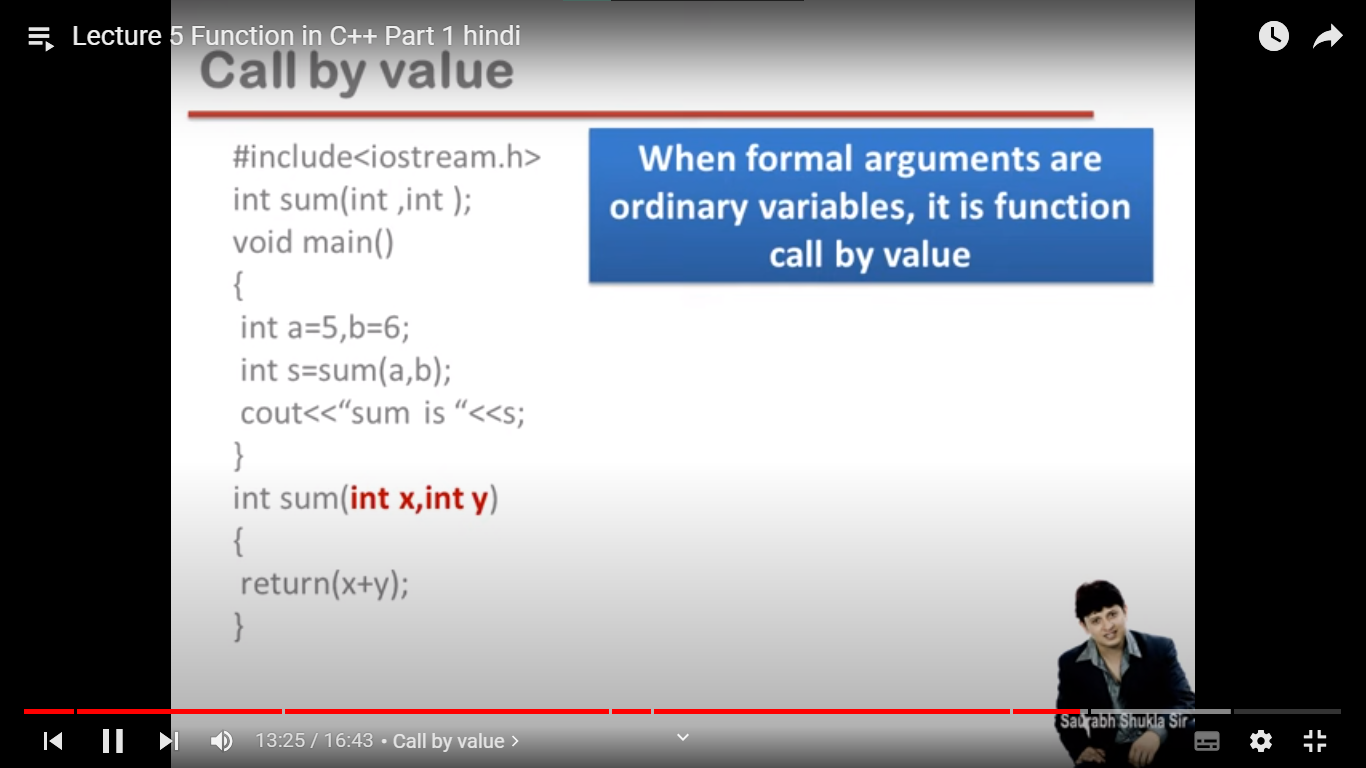
cout << "Calling globalFunction() from localFunction():\n";

globalFunction(); // A global function can be called from a local function but vice versa might not be possible

}\*/

Call by value:

->when formal arguments are ordinary variables then it is called as call by value



Call by Address:

->when formal arguments are pointer variables, it is called as call by address

int sum(int \*a, int \*b){

return (\*a+\*b);

}

int main(){

cout<<sum(&a,&b)<<endl;

}

#include <iostream>

// Function that takes a pointer to an int and modifies the value it points to

void modifyValue(int\* ptr) {

\*ptr = 100; // Change the value pointed to by ptr to 100

}

int main() {

int num = 42;

std::cout << "Original value: " << num << std::endl; // Output: Original value: 42

// Pass the address of 'num' to the function 'modifyValue'

modifyValue(&num);

std::cout << "Modified value: " << num << std::endl; // Output: Modified value: 100

return 0;

}

Original value gets modified!\

Call by reference:

->when formal arguments are reference variable then it is called as call by reference.

Memory is same but can have different names.

int sum(int &a, int &b){

return (a+b);

}

int main(){

cout<<sum(a,b)<<endl;

}

Original value gets modified!!

->adv:(saves memory space)Function in a program is to save memory space which becomes appreciable when a function is likely to be called many times.

->disadv(time consuming):However every time a function is called, it takes lot of extra time in executing a series of instructions for tasks such as jumping to functions, pushing arguments to stacks.

-> so when function is small it is worthless to spend so much extra time in such tasks in cost of saving comparatively small space.

-> so there is a problem in making small functions.

\*Inline Function

->To eliminate the cost of calls to small functions, c++ proposes a new feature called inline function.

->An inline function is a function that is expanded in line when it is invoked.

->Compiler replaces the function call with the corresponding function code.

->Inline is a request not a command

-> Compiler gets to decide based on memory usage whether to accept a function as inline or treat it as a normal function.

-> If function has loops, switch, go to, recursion, static variables, etc compiler does not accept it as inline.

inline int square(int x);

int main() {

int a = 5;

int sq = square(a);

cout << "Square of " << a << " is: " << sq << endl;

}

int square(int x) { return x \* x; }

\*Default arguments:

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn’t provide a value for the argument. In case any value is passed, the default value is overridden.

// Assigning default values starts from right side

int add(int x, int y, int z = 0) {

return x + y + z;

}

int main() {

int a = 2, b = 3, c = 5;

cout << "Addition of " << a << " and " << b << " is: " << add(a, b) << endl; // 5

cout << "Addition of " << a << ", " << b << " and " << c << " is: " << add(a, b, c) << endl; //10

}

#include<bits/stdc++.h>

using namespace std;

// Assigning default values starts from right side

int add(int, int, int = 0);

int main() {

int a = 2, b = 3, c = 5;

cout << "Addition of " << a << " and " << b << " is: " << add(a, b);

cout << "Addition of " << a << ", " << b << " and " << c << " is: " << add(a, b, c);

}

int add(int x, int y, int z) {

return x + y + z;

}

Function Overloading:

In C, we could not have more than one function having the same name

In C++ we can do so using function overloading

🡪 More than one function with same name and difference in arguments (and they need to follow some rules obviously

🡪 Compile time polymorphism

IMP:

int area(int,int);

float area(int,int);

This will give error. Type or count of arguments has to be different

BUT, int area(int);

float area(int,int);

This will not give error. Type or count of arguments are different

One of the important tasks of compiler is to bind function call with the actual function definition. This is called Early Binding.

Steps:

1. When the compiler reaches a function call line, it checks the function name of the function call.

2. All the functions defined in the program with the same name are candidates for being binded.

3. Based on some rules it then checks which function to bind

Rules:

1. The compiler tries to find exact match to function call (same number and type of arguments)

2. If no match is found, it tries to find match by promotion:

char, unsigned char, short --> Promoted to int

float --> Promoted to double

3. If, no match is found, it tries to find match via standard conversion

If the function call does not fit in any of the rules, then an error occurs.

Example of functional overloading:

int sum(int x, int y) {

return x + y;

}

int sum(int x, int y, int z) {

return x + y + z;

}

int main() {

int a = 2, b = 3, c = 5;

// Follows rule 1

cout << "Sum of " << a << " and " << b << " is: " << sum(a, b);

cout << "Sum of " << a << ", " << b << " and " << c << " is: " << sum(a, b, c);

// Follows rule 2

cout << "Sum of 'a' & 'b' is:(Prints sum of ASCII) " << sum('a', 'b');

// Follows rule 3

cout << "Sum of 1.1 & 2.9 is:(Prints sum of floor values) " << sum(1.1, 2.9) ;

}

Output:

Sum of 2 and 3 is: 5

Sum of 2, 3 and 5 is: 10

Sum of 'a' & 'b' is:(Prints sum of ASCII) 195

Sum of 1.1 & 2.9 is:(Prints sum of floor values) 3

What is structure?

--> Collection of dissimilar elements

--> Way to group variables

--> Used to create user-defined Data Type

--> Memory is only assigned when structure variable/instance of struct is created

and not when struct is defined.

--> Structure can be created globally and locally inside the main function also. if it is created inside main function then it only can be accessed inside the main function and if it is created globally then it can be accessed anywhere in the program.

-> IMP: Writing the keyword struct before declaring a new structure variable is optional in C++ but mandatory in C

Some ways to create and initialize structure variable -->

// Ways of declaration

\*\* First Type \*\*

struct myStruct(){

// member variables

} s1;

\*\* Second Type \*\*

myStruct s2={initialize here};

\*\* Third Type \*\*

myStruct s3;

type3={initialize here}

\*\* Fourth Type \*\*

myStruct s4;

type4.someMemberVariable=someValue (Use of '.' operator)

\*\* Fifth Type \*\*

myStruct type5;

type5=type3 (Initialize with any other struct variable directly)

struct Point {

int x = 1; // Can initialize with some value in C++ but not in C

int y = 1;

} p1;

Point p2 = {2, 3};

Point p3;

p3 = {4, 5};

Point p4;

p4.x = 6, p4.y = 7;

struct Employee {

private:

int eID;

string name;

int age;

public:

// Structs can also have constructors

Employee() {

eID = 0;

name = "";

age = 0;

}

void getInputData() {

cout << "Enter Employee ID, Employee name & Employee age: ";

/\*

No Need of '.' operator to access member variables. (It will actually give CE)

This (->) pointer can be used but no need as such

\*/

cin >> eID >> this->name >> age;

}

void printData(); // You can also define functions outside the struct

};

void Employee::printData() {

cout << "Hello " << name << ". Your Employee ID is " << eID << " and your age is " << age << endl;

}

int main() {

/\*

In C, structs could not have member functions but they can have in C++.

This is the beginning step to achieve encapsulation

(See Struct Employee)

IMP:

1. Struct can also have access specifiers public, private and protected

(See Employee struct)

\*/

Employee e1;

e1.getInputData();

e1.printData();

Employee e2;

e2.printData(); // Values set in constructor will get printed

}

Difference of struct in c and c++:

1. Writing the keyword struct before declaring a new structure variable is optional in C++ but mandatory in C
2. In C, structs could not have member functions but they can have in c++
3. Struct in c does not have access specifiers, c++ does have.

Differences between classes and structures:

1. Access specifiers are public for structs by default and private for classes

2. Inheritance defaults to public for structs by default and private for classes

Except these, you can do everything in struct and class in similar fashion.

\*Procedure - Oriented Programming

-->The problem is viewed as a sequence of things to be done and broken down into subproblems. Each problem is dealt with instructions which are organized into groups known as functions.

Features:

Most of the functions share global data.

Data moves openly from one function to another.

Employs top-down approach in program design.

Disadvantages:

Emphasis is on functions.

Global data is vulnerable to an accidental change by a function.

In case the external data structure is being changed, all functions have to be revised.

\*Object-Oriented Programming

-->OOP is a paradigm where everything is revolved around objects. OOP decomposes the problem into several entities called objects and builds data and functions around these objects.

-->Emphasis is on data rather than procedure.

-->Objects may communicate with each other through functions.

-->Follows bottom-up approach in program design.

Need of oop

->Modularity and Code Organization

->Code Reusability

->Encapsulation and Data Hiding

\*Class

-->A class is blueprint of object.

-->A Class is a user defined data-type which has data members and member functions. Variable is a block which contains single data, and object is data type

Which can store multiple variable and functions.

-->Class is a user-defined data type of which objects are variable.

🡪Class is description of object (Basically blueprint).

🡪It is used to describe properties and behavior of an object.

🡪Object is an instance of a class.

🡪Class is a means to achieve encapsulation.

🡪Object is a runtime entity, class does not have any memory object has, it depends on class that how much memory is allocated to the object.

🡪Creating a class as good as defining a new data type.

-->For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties. The data member will be speed limit, mileage etc. and member functions can be applying brakes, increase speed etc.

Differences between classes and structures:

1. Access specifiers are public for structs by default and private for class

2. Inheritance defaults to public for structs by default and private for classes

Except these, you can do everything in struct and class in similar fashion

\*Object:

-->An Object is a real-world entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

-->There are 2 ways to define a member function:

Inside class definition

Outside class definition

To define a member function outside the class definition we have to use the scope resolution:: operator along with class name and fun ction name.

Difference between defining a function inside the class and outside the class:

--> Functions defined inside the class are inline by default but when they are defined outside the class, inline keyword has to be used explicitly to request for inline functionality

IMP: State of an object should only be changed by instance member functions. State here means values of member variables.

Different names for Instance member variables: Attributes, data members, fields, properties, etc.

Different names for Instance member functions: Methods, procedures, actions, operations, services, etc.

#include<bits/stdc++.h>

using namespace std;

class Point

{

private:

    int x, y; // Called as instance member variables mhnje ki class che nahiyet te

              // object che ahet teee

public:

    // instance member functions

    void setData(int x, int y)

    {

        this->x = x;

        this->y = y;

    }

    void getData();

    int getX()

    {

        return x;

    }

    int getY()

    {

        return y;

    }

    // Manhattan distance between 2 points

    int manhattanDistance(Point p)

    {

        return abs(x - p.x) + abs(y - p.y);

    }

};

/\*

Point: --> This thing is called membership label (Class name and then scope resolution operator)

\*/

// Define member function outside the class

void Point::getData()

{

    cout << "The coordinates of point are: (" << x << ", " << y << ")" << endl;

}

int main()

{

    // Point is the data type and p1 is the variable, but in oop we called it as     // // object

    Point p1;

    p1.setData(2, 3);

    p1.getData();

    Point p2;

    p2.setData(4, 5);

    p2.getData();

    int distance = p1.manhattanDistance(p2);

    cout << "Manhattan distance between (" << p1.getX() << " ," << p1.getY() << ") and (" << p2.getX() << ", " << p2.getY() << ") is " << distance;

}

**\*Access specifiers**

-->Access modifiers are used to implement an important aspect of Object-Oriented Programming known as Data Hiding.

There are 3 types of access modifiers available in C++:

**Public**

**Private**

**Protected**

If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be Private.

1. **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.

2. **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. Only the member functions or the friend functions are allowed to access the private data members of the class.

However, we can access the private data members of a class indirectly using the public member functions of the class. Getter and setter’s

class BankAccount

{

private:

    string accountNumber;

    double balance;

public:

    BankAccount(string accNumber, double initialBalance){

        this->accountNumber = accNumber;

        this->balance = initialBalance;

    }

    string getAccountNumber()

    {

        return accountNumber;

    }

};

int main()

{

    // Create an instance of the BankAccount class

    BankAccount account("123456789", 1000.0);

    // Access private data indirectly using public member functions

    std::cout << account.getAccountNumber() << std::endl; // Output: 123456

}

3. Protected: The protected access modifier is similar to the private access modifier in the sense that it can’t be accessed outside of its class unless with the help of a 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.

**Static local variables: (Variable of a class!)**

🡪 Concept is taken from C

🡪 By default, initialized to 0 and not some garbage value.

🡪 Only one copy is present throughout the lifecycle of the program.

--> Can be used to implement co-routines in C/C++

void fun(){

static int x;

x++;

cout<<x<<endl; // 1

int y;

}

when we call fun function then only y variable will get memory but x variable get memory at the start of program.

Static Member Variables:

--> Declared inside the class body

🡪 Belongs to class, not the object!

--> Also known as class member variables

--> must be defined outside the class (MANDATORY)

--> They do not belong to any object but belong to the entire class.

--> Only one copy of static member variable for the whole class throughout the runtime of the program.

--> They can also be used with class name.

--> Memory for static member variable only gets allocated after member definition and not after declaration.

🡪 It is created even if the object is not created!

Since static member variable is class variable, it can be accessed without an object

If it is a public member it can directly be accessed using className::staticVariableName

But if it is private/ protected we need to have some functions to get and set the values.

But if the function is a member function, then it can only be accessed using objects.

But we should be able to access static member variables without objects also.

Thus, static member functions come into picture

Static Member Functions: (class member function!)

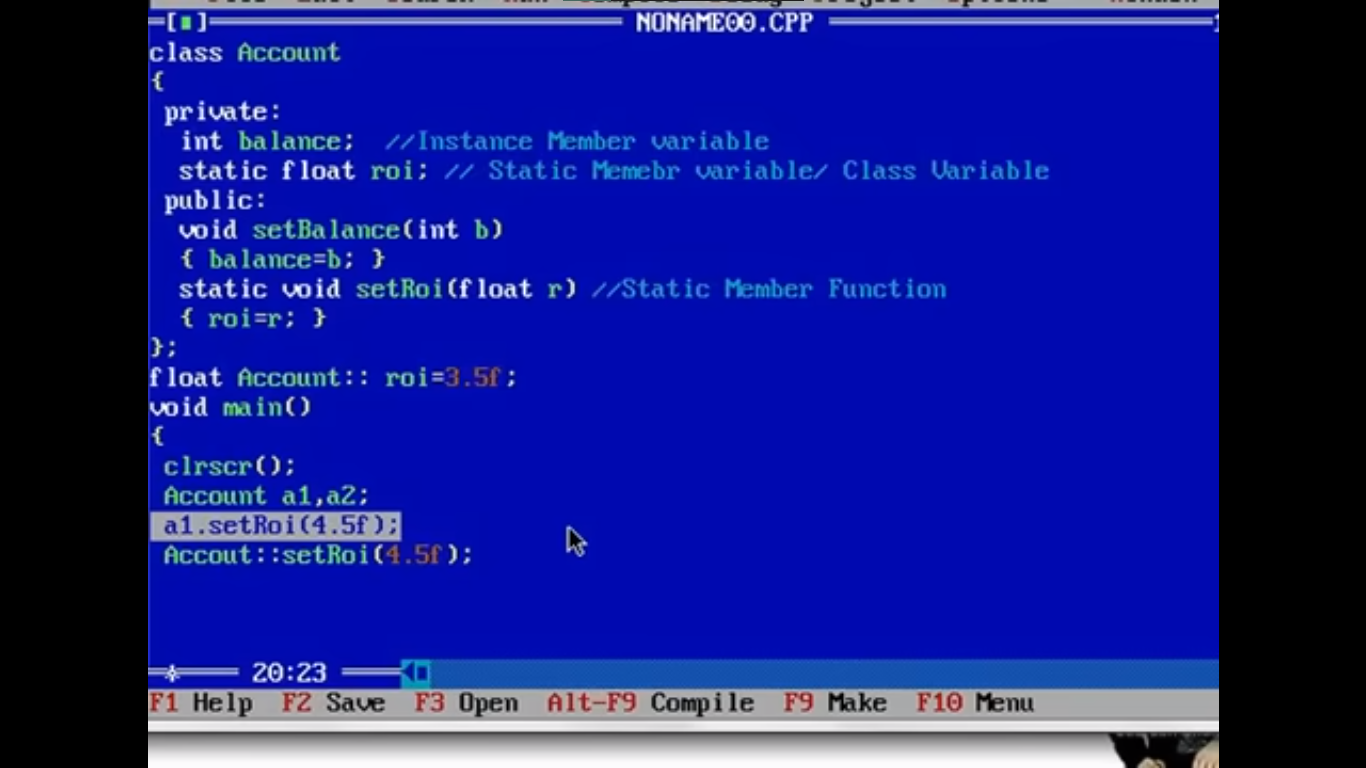
--> Can only access static member variables

🡪 Used to assign/set values to static variable which are private

--> Can also be invoked without an object

🡪 setRoi is the static member function, to set the value of static variable ‘roi’

Which is private. And is accessed in main function without creation of any object.



class Account {

int balance; // This is the variable of the object, not of the class as this is of

// no use till it is accessed by the object/instance of class

static float rateOfInterest; // Static member variable or Class variable

public:

void setBalance(int balance) {

this->balance = balance;

}

int getBalance() {

return balance;

}

static float getRateOfInterest() { // Static member function!

return rateOfInterest;

}

/\*

We cannot use this pointer (->) with static member functions

Example:

this->rateOfInterest=rateOfInterest is not correct and causes CE

\*/

static void setRateOfInterest(float r) { // Static member function or Class member function

rateOfInterest = r;

}

};

// The line below is extremely important for static member variables.

// Without this line an error will occur and we cannot access the value present // in rateOfInterest

// Error if the line below is not present: undefined reference to //`Account::rateOfInterest'

// We can also set some default value for the static variable. It is 0 by default

The error "undefined reference to **Account::rateOfInterest'" occurs when the linker cannot find the definition of a static member variable** rateOfInterest**that is declared in the**Account` class. This error is related to how static member variables are defined and used in C++.

When you declare a static member variable in a class, you need to provide its definition in one of the source files. This ensures that the memory for the static variable is allocated and initialized properly. If the definition is missing or not accessible during the linking phase, you will encounter the "undefined reference" error.

// Definition and initialization of the static member variable

float Account::rateOfInterest = 6.5f; // This line is extremely important! It initializes a static variable!!

int main() {

Account a1;

a1.setBalance(200);

cout << a1.getBalance() << endl;

cout << Account::getRateOfInterest() << endl;

}

\*Constructors:

--> The work of constructor is to make an object an object (Tough to understand statement:) )

--> Instance Member function of class (Hence, can never be static)

--> Name of constructor is same as class name

--> Cannot use return keyword

--> It is implicitly invoked when an object is created.

--> Used to solve the problem of initialization of member variables. i.e. members of class will be initialized!

--> If any object is created, a constructor has to be invoked (No questions asked it has to be invoked)

There are 3 types of constructors:

**Default constructors**

**Parameterized constructors**

**Copy constructors**

Constructor has same name as the class itself

Default Constructors don’t have input argument however, Copy and Parameterized Constructors have input arguments

Constructors don’t have return type

A constructor is automatically called when an object is created.

It must be placed in public section of class.

If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

let us understand the types of constructors in C++ by taking a real-world example. Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options. The first one you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn’t mention anything just say you want a marker. So when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is! The second method is you go to a shop and say I want a marker a red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is! Then the third one you go to a shop and say I want a marker like this (a physical marker on your hand). So, the shopkeeper will see that marker. Okay, and he will give a new marker for you. So, copy of that marker. And that’s what a copy constructor is!

Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.

Constructors do not return values; hence they do not have a return type.

A constructor gets called automatically when we create the object of the class.

Constructors can be overloaded. i.e. Multiple Constructor can be created with changing different parameters.

Constructor cannot be declared virtual.

**How to use Constructors in private section?**

Using Friend Class : If we want that class should not be instantiated by anyone else but only by a friend class.

class A{

private:

A(){

cout << "constructor of A\n";

}

friend class B;

};

// class B, friend of class A

class B{

public:

B(){

A a1;

cout << "constructor of B\n";

}

};

// Driver program

int main(){

B b1;

return 0;

}

Output:

constructor of A

constructor of B

\*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.

However, constructors in C++ are not virtual. Constructors are automatically called when an object is created and are responsible for initializing the object's data members and setting up its state. Constructors are not inherited, and derived classes do not override base class constructors. When you create an object of a derived class, the base class constructor is called first, and then the derived class constructor is called.

In C++, virtual constructors are not supported because constructors are not like regular member functions. Constructors are responsible for initializing the object's data members and setting up its state during object creation, and they have specific rules and behaviors that cannot be achieved using virtual functions.

A virtual function in C++ is a base class member function that you can redefine in a derived class to achieve polymorphism.

--> Virtual function is possible in c++

1. Default Constructors: Default constructor is the constructor which doesn’t take any argument. It has no parameters. It is also called a zero-argument constructor. Default constructor’s which are created by class itself when no constructor is defined explicitly, are of two types: Default constructor( without arguments vala) and a Copy constructor(when object is created by passing another object as a parameter then here comes the picture of shallow copy and deep copy!)
2. Parameterized Constructors: a constructor which takes argument

when the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as Student s; Will flash an error

-->Uses of Parameterized constructor:

It is used to initialize the various data elements of different objects with different values when they are created.

It is used to overload constructors.

Can we have more than one constructor in a class?

Yes, It is called Constructor Overloading.

-->In C++, we can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading

-->While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

3. Copy Constructor:

-->A copy constructor is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor.

-->The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

// Copy constructor

Point(const Point& p1)

{

x = p1.x;

y = p1.y;

}

-->The default constructor does only shallow copy.

-->Deep copy is possible only with a user-defined copy constructor. In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new memory locations. i.e. both the objects have different memory resources.

-->The main difference between Copy Constructor and Assignment Operator is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

MyClass t1, t2;

MyClass t3 = t1; // ----> (1) default copy constructor

t2 = t1; //-->(2) default assignment operator(this is not copy constructor)

Copy donhi way ne hot ahe, pn ways different ahet. Donhi shallow copies ahet mhanu shakto tu, pan jar apan modified copy constructor lihila tr deep copy hote.

Jevha object madhe pointers ahet tevha deep copy zali pahije!! Tyasathi aaplyala swatah copy constructor/copy assignment operator banvava lagta.

https://www.youtube.com/watch?v=nCAVr\_T4DbM&list=PLLYz8uHU480j37APNXBdPz7YzAi4XlQUF&index=55&ab\_channel=C%2B%2BbySaurabhShuklaSir

the copy constructor is used for creating new independent copies of objects, while the assignment operator is used to modify an existing object with the content of another object.

-->Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So, if we pass an argument by value in a copy constructor, a call to the copy constructor would be made to call the copy constructor which becomes a non-terminating chain of calls. Therefore, compiler doesn’t allow parameters to be passed by value.

-->Why argument to a copy constructor should be const?

One reason for passing const reference is, that we should use const in C++ wherever possible so that objects are not accidentally modified.

Some important points:

--> By default, when a programmer does not create any constructor by ownself, the compiler creates a default constructor and a copy constructor.

--> When a programmer creates a copy constructor by ownself, the compiler does not create any constructors.

--> When a programmer creates a default constructor by ownself, the compiler creates a copy constructor.

🡪 The compiler doesn’t create a default constructor if we write any constructor even if it is a copy constructor.

**What happens when we write a normal constructor and don’t write a copy constructor?**

The compiler creates a copy constructor if we don’t write our own. The compiler creates it even if we have written other constructors in a class.

private:

int x, y;

public:

Point() { // Default Constructor

x = 0, y = 0;

}

Point(int k) { // Parameterized Constructor

x = k, y = k;

}

Point(int x, int y) { // Parameterized Constructor

this->x = x;

this->y = y;

}

Point(Point &p) { // Copy Constructor

this->x = p.x;

this->y = p.y;

}

};

int main() {

// Different ways of using constructors to initialize an object

Point p1;

Point p2(1, 2);

Point p3(1);

/\*

The p4 declaration gives CE only when user defines a copy constructor by ownself. If no explicit copy constructor is present the below line is syntactically and logically correct

\*/

// Point p4 = 1;

Point p5 = p3;

Point p6(p3);

}

#include<bits/stdc++.h>

using namespace std;

class Point {

private:

    int x, y;

public:

    // Point() { // Default Constructor

    //  x = 0, y = 0;

    // }

    // Point(int k) {  // Parameterized Constructor

    //  x = k, y = k;

    // }

    // Point(int x, int y) {   // Parameterized Constructor

    //  this->x = x;

    //  this->y = y;

    // }

    Point(Point &p) {   // Copy Constructor

        this->x = p.x;

        this->y = p.y;

    }

};

int main() {

    /\*

        What is a constructor?

            --> The work of constructor is to make an object an object (Tough to understand statement:) )

            --> Instance Member function of class (Hence, can never be static)

            --> Name of constructor is same as class name

            --> Cannot use return keyword

            --> It is implicitely invoked when an object is created

            --> Used to solve the problem of intialization of member variables

            --> If any object is created, a constructor has to be invoked (No questions asked it has to be invoked)

        Some important points:

            --> By default when a programmer does not create any constructor by ownself, the compiler creates

                a default constructor and a copy constructor

            --> When a programmer creates a copy constructor by ownself, the compiler does not create any constructors

            --> When a programmer creates a default constructor by ownself, the compiler creates a copy constructor

        Some things wrt Copy Constructor:

            Don't you feel the syntax for copy constructor would be more intutive if it were:

                ClassName (Classname Object){

                    // Copy stuff

                }

            But why an additional & sign is needed before the object?

            So, if the syntax were without the & sign, Clasname Object would call the copy constructor recursively for

            copying the actual arguments

            Thus, a reference is passed

    \*/

    // Different ways of using constructors to initialize an object

    Point p1;

    Point p2(1, 2);

    Point p3(1);

    /\*

        The p4 declaration gives CE only when user defines a copy constructor by ownself. If no explicit copy

        constructor is present the below line is syntactically and logically correct

    \*/

    // Point p4 = 1;

    Point p5 = p3;

    Point p6(p3);

}

As I have created just copy constructor, so compiler will not create the default constructor, so object p1 will not be created!! For that we need to create default constructor explicitely!!

Errors while compiling:

e:\CPP\treeHard.cpp: In function 'int main()':

e:\CPP\treeHard.cpp:63:8: error: no matching function for call to 'Point::Point()'

  Point p1;

        ^~

e:\CPP\treeHard.cpp:25:2: note: candidate: Point::Point(Point&)

  Point(Point &p) {   // Copy Constructor

  ^~~~~

e:\CPP\treeHard.cpp:25:2: note:   candidate expects 1 argument, 0 provided

e:\CPP\treeHard.cpp:64:15: error: no matching function for call to 'Point::Point(int, int)'

  Point p2(1, 2);

               ^

e:\CPP\treeHard.cpp:25:2: note: candidate: Point::Point(Point&)

  Point(Point &p) {   // Copy Constructor

  ^~~~~

e:\CPP\treeHard.cpp:25:2: note:   candidate expects 1 argument, 2 provided

e:\CPP\treeHard.cpp:65:12: error: no matching function for call to 'Point::Point(int)'

  Point p3(1);

            ^

e:\CPP\treeHard.cpp:25:2: note: candidate: Point::Point(Point&)

  Point(Point &p) {   // Copy Constructor

  ^~~~~

e:\CPP\treeHard.cpp:25:2: note:   no known conversion for argument 1 from 'int' to 'Point&'

/\*

How to create copy of an object?

--> Copy Constructor

--> Implicit copy assignment operator/Default assignment operator

Shallow Copy:

--> Creating copy of an object by copying data of all member variables as it is is called shallow copy

\*/

class ShallowCopy {

private:

int x, y;

public:

ShallowCopy() {

x = 0, y = 0;

}

void setData(int x, int y) {

this->x = x, this->y = y;

}

void getData() {

cout << "X: " << x << ", Y: " << y << endl;

}

ShallowCopy(ShallowCopy &c) {

x = c.x, y = c.y;

}

};

int main() {

ShallowCopy shallow1;

shallow1.setData(2, 3);

ShallowCopy shallow2;

shallow2 = shallow1;

shallow2.getData();

}

Deep Copy:

--> Creating an object by copying data of another object along with the values of memory resources which resides outside the object but are handled by the object.

class DeepCopy {

private:

int x, y;

int\* p;

public:

DeepCopy() {

p = new int;

}

void setData(int x, int y, int z) {

this->x = x;

this->y = y;

\*p = z;

}

void getData() {

cout << "X: " << x << ", Y: " << y << ", \*P: " << \*p << endl;

}

/\*

Why is deep copy required?

--> In this class p is a pointer variable which will hold the address of an integer memory block

--> So if we do normal copy as we do above, contents of p will be copied to new object.

--> This implies that the address of that integer variable (stored in p) will be copied.

--> So p from 2 different objects will have same \*p

--> Hence, we need to declare a new int at runtime and then do a deep copy

\*/

DeepCopy(DeepCopy &c) {

x = c.x, y = c.y;

p = new int;

\*(p) = \*(c.p);

}

/\*

Dangling pointer stuff:

--> So if the lifetime of an object is over and we do not explicitely delete the existance

of \*p, it will be there in the memory with no access point.

--> It can cause unnecessary crashes and RTE.

--> So, deleting via a destructor is a good practice.

\*/

~DeepCopy() {

delete p;

}

};

int main(){

DeepCopy deep1;

deep1.setData(4, 5, 6);

DeepCopy deep2;

deep2 = deep1;

deep2.getData();

}

#include <bits/stdc++.h>

using namespace std;

class ShallowCopy

{

private:

    int x, y;

public:

    ShallowCopy()

    {

        x = 0, y = 0;

    }

    void setData(int x, int y)

    {

        this->x = x, this->y = y;

    }

    void getData()

    {

        cout << "X: " << x << ", Y: " << y << endl;

    }

    ShallowCopy(ShallowCopy &c)

    {

        x = c.x, y = c.y;

    }

};

class DeepCopy

{

private:

    int x, y;

    int \*p;

public:

    DeepCopy()

    {

        p = new int;

    }

    void setData(int x, int y, int z)

    {

        this->x = x;

        this->y = y;

        \*p = z;

    }

    void getData()

    {

        cout << "X: " << x << ", Y: " << y << ", \*P: " << \*p << endl;

    }

    /\*

        Why is deep copy required?

            --> In this class p is a pointer variable which will hold the address

of an integer

                memory block

            --> So if we do normal copy as we do above, contents of p will be

copied to new object.

            --> This implies that the address of that integer variable (stored in

p) will be copied.

            --> So p from 2 different objects will have same \*p

            --> Hence, we need to declare a new int at runtime and then do a deep

copy

    \*/

    DeepCopy(DeepCopy &c)

    {

        x = c.x, y = c.y;

        p = new int;

        \*(p) = \*(c.p);

    }

    /\*

        Dangling pointer stuff:

            --> So if the lifetime of an object is over and we do not explicitely delete the existance

                of \*p, it will be there in the memory with no access point.

            --> It can cause unnecessary crashes and RTE.

            --> So, deleting via a destructor is a good practice.

    \*/

    ~DeepCopy()

    {

        delete p;

    }

};

int main()

{

    /\*

        How to create copy of an object?

        --> Copy Constructor

        --> Implicit copy assignment operator

        Shallow Copy:

            --> Creating copy of an object by copying data of all member

variables as it is is called shallow copy

        Deep Copy:

            --> Creating an object by copying data of another object along with

the values of memory resources which resides outside the object

but are handled by the object.

    \*/

    ShallowCopy shallow1;

    shallow1.setData(2, 3);

    ShallowCopy shallow2;

    shallow2 = shallow1;

    shallow2.getData(); // sallpo madhe ahe tasa copy hoel sagla !

    DeepCopy deep1;

    deep1.setData(4, 5, 6);

    DeepCopy deep2 = deep1;

    // by above line created copy constructor will be invoked, and both will be

Separate objects now, both will contain separate block for p this will

create actually deep copy !!

    // deep2 = deep1;

    // by above line Copy constructor will not be invoked, default assignment

operator invoke hoel je ki shallow copy create karel mhnje ahe tasa sagla

copy karel jar apan aapla assignment operator lihila asta tar deep copy

create zali asti ani donhi separate aste, pan ithe shallow copy create

hoel jyane deep1 cha p jya block la address karat ahe tyach block la deep2

cha p point karel ani jar ekane update kela tr te doghi kade update hoel,

pn jar copy constructor ne jar object copy kela tr deep copy create hoel

ani donhi object separate p vala block thevtil ekane update kela tr

dusryala farak nahi padnar.

    deep1.setData(4, 5, 9);

    deep1.getData();

    deep2.getData();

}

Output:

X: 2, Y: 3

X: 4, Y: 5, \*P: 9

X: 4, Y: 5, \*P: 6

\*Destructor

#include<bits/stdc++.h>

using namespace std;

class Point {

private:

    int x, y;

public:

    Point() {

        x = 0, y = 0;

        cout << "Default constructor invoked\n";

    }

    Point(int x, int y) {

        this->x = x;

        this->y = y;

        cout << "Parametrized constructor invoked\n";

    }

    ~Point() {

        cout << "Destructor invoked\n";

    }

};

int main() {

    /\*

        What is a destructor?

            --> It is an instance member function of a class hence can never be

static

            --> Destructor name is same as class name preceded by ~(pronounced

tilde) symbol

            --> No return type

            --> No arguments, hence no overloading is possible

            --> It is invoked implicitly when object is going to get destroyed

        Use of destructor?

            --> It should be defined to release resources allocated to an object

        Some important points:

            --> If a programmer does not explicitly create a destructor, compiler

creates one.

            --> But if the programmer creates one explicitly, the compiler does

not create one.

    \*/

    Point p1;

    Point p2(2, 3);

}

Output:

Default constructor invoked

Parametrized constructor invoked

Destructor invoked

Destructor invoked

class Complex {

private:

int x, y;

public:

Complex() {

x = 0, y = 0;

}

Complex(int x, int y) {

this->x = x, this->y = y;

}

void getData() {

if (y >= 0)

cout << "(" << x << "+" << y << "i)\n";

else

cout << "(" << x << "" << y << "i)\n";

}

void setData(int x, int y) {

this->x = x, this->y = y;

}

Complex operator +(Complex &c) {

Complex temp;

temp.x = x + c.x;

temp.y = y + c.y;

return temp;

}

// Overloading unary operators

Complex operator -() {

Complex temp;

temp.x = -x;

temp.y = -y;

return temp;

}

// Overloading pre-increment operator

Complex operator ++() {

Complex temp;

temp.x = ++this->x;

temp.y = ++this->y;

return temp;

}

/\*

Overloading post-increment operator

Why int is present as an argument?

--> This is done to help the compiler differentiate between pre and post increment operators after

being overloaded

--> Only 'int' should be written. Nothing else is acceptable (Else, this error occurs: no ‘operator++(int)’ declared for postfix ‘++’);

\*/

Complex operator ++(int) {

Complex temp;

temp.x = this->x++;

temp.y = this->y++;

return temp;

}

};

int main() {

/\*

Operator Overloading:

--> When an operator is overloaded with multiple jobs, it is known as operator overloading

--> Way to implement compile time polymorphism

You cannot overload:

. (dot)

::

?:

sizeof

\*/

Complex c1(2, 3), c2(3, 4), c3, c4, c5, c6, c7, c8;

cout << "First complex number is: "; c1.getData();

cout << "Second complex number is: "; c2.getData();

// c1 will be the called object. While overloading binary operator, left object will be the caller object

c3 = c1 + c2;

cout << "Addition of first and second complex numbers is: "; c3.getData();

c4 = c1.operator + (c2);

cout << "Addition of first and second complex numbers(Different syntax) is: "; c4.getData();

c5 = -c1;

cout << "Negative of first complex number is: "; c5.getData();

c6 = c1.operator - ();

cout << "Negative of first complex number(Different Syntax) is: "; c6.getData();

c7 = c1++;

cout << "Result after Post increment operator overloading on first complex number is: "; c7.getData();

cout << "After post incrememnt first complex number becomes: "; c1.getData();

c8 = ++c2;

cout << "Result after Pre increment operator overloading on second complex number is: "; c8.getData();

cout << "After pre incrememnt second complex number becomes: "; c2.getData();

}

class Breadth;

class Length {

private:

int l;

public:

friend int area(Length, Breadth);

void setL(int l) {

this->l = l;

}

int getL() {

return l;

}

};

class Breadth {

private:

int b;

friend int area(Length, Breadth);

public:

void setB(int b) {

this->b = b;

}

int getB() {

return b;

}

};

int area(Length l1, Breadth b1) {

return l1.l \* b1.b;

}

int main() {

/\*

What is friend function?

--> Friend function is not a member function of a class to which it is a friend

--> It should be declared inside the class

--> It must be defined outside the class of which it is a friend

--> Since it is a friend to a class, it can access any of the members of that class

to which it is a friend, but not directly

--> It has no caller object

--> It should not be defined with membership label

--> It can be a friend to more than one class

--> You can declare a friend function anywhere in the class, access specifiers do not matter since it is a friend

\*/

Length l;

Breadth b;

l.setL(2);

b.setB(3);

cout << area(l, b) << endl;

}

// Predecleration of class B is necessary becuase it is being passed as an argument in function1

class B;

class A {

private:

int x = 2;

public:

/\*

If it is to be used as a friend to some other class, it should be defined after that class's definition

and not int class A.

\*/

void function1(B);

/\*

It means that C (to be specific all member functions of C)can access all the private members of A

\*/

friend class C;

};

class B {

private:

int x = 0;

/\*

This statement implies that function1 from class A is a friend of class B

and can access private members of class B

\*/

friend void A::function1(B);

};

/\*

Definition of the member function from class A which is friend to class B

\*/

void A::function1(B b) {

cout << "In Function 1 of class A\n";

cout << "Accessing value of x(Private member of B) outside class B: b.x = " << b.x << endl;

}

class C {

public:

void showA(A a) {

cout << "In showA function of class C\n";

cout << "Accessing value of x(Private member of A) outside class A: a.x = " << a.x << endl;

}

};

int main() {

A a;

B b;

C c;

a.function1(b);

c.showA(a);

}

Merits:

A friend function is able to access members without the need of inheriting the class.

Friend function acts as a bridge between two classes by accessing their private data.

It can be used to increase the versatility of overloaded operator.

It can be declared either in the public or private or protected part of class.

Demerits:-

Friend functions have access to private members of a class from outside the class which violates the law of the data hiding.

Friend functions cannot do any run time polymorphism in its members.

class A {

private:

int x;

protected:

void setX(int x) {

this->x = x;

}

int getX() {

return x;

}

};

class B: public A {

public:

void setData(int x) {

setX(x);

}

int getData() {

return getX();

}

};

int main() {

/\*

What is inheritance?

--> It is a process of inheriting properties and behaviours of existing class(es) into new class(es)

--> A more generalized class is always the parent and the specific one is always the child

Example:

Class Fruit will be a parent class

Class Apple will be a child class

Need of inheritance?

--> Let us understand this via and example. Suppose you have a class Car. You also wish to have a class

SportsCar which has all the properties of Car + some exclusive properties for it being a SportsCar

--> What we can do for this?

--> We can have 2 seperate classes, Car (With original properties)

and SportsCar (With the new properties + properties of car). Issue in this is that we will have to write

same code twice for Car and SportsCar for the common features

--> We can have 2 seperate classes, Car (With original properties) and SportsCar (With the new properties)

and then whenever we need to add a new SportsCar we make 2 objects for it. One for new features of a Sports

Car and general features of a Car. Issue with this is that it does not comply with the idea of Encapsulation

which states that for any entity all of it's variables and functions should be part of the same class

So to solve these problems, the concept of inheritance comes into picture.

We can reuse the properties of a previously declared class and use them in our new class with some

additional functionalities

Syntax:

class BaseClassName{

};

class DerivedClassName : VisibilityMode/AccessSpecifier BaseClassName{

};

Types of Inheritance:

1. Single Inheritance

--> 1 parent class and 1 child class

Example:

Class A{

};

class B:public A{

};

2. Multilevel Inheritance

--> More than one level of single inheritance

Example:

Class A{

};

Class B: public A{

};

class C: public B{

};

3. Multiple Inheritance

--> More than one parent class

Example:

Class A{

};

Class B{

};

Class C: public A, public B{

};

4. Hierarchical Inheritance

--> One parent having more than one children

Example:

Class A{

};

Class B: public A{

};

Class C: public A{

};

5. Hybrid Inheritance

--> Mixture of more than one type of inheritance from 1 to 4

Visibility Modes

--> Private

--> Protected

--> Public

Types of users for a class

--> Users who create an object of the class (say u1)

--> Users who use derived class to access members of base class (say u2)

Difference between availability & accessibility?

--> During inheritance everything is inherited may it be Private or Protected or Public.

They are avaiable in the derived class and availalibility does not depend on the visibility mode

of inheritance.

When we say they are available, it does not mean that everyone can access it. They might not

be accessible depending on the visibility mode

Access specifier in base class | Access specifier when inherited publicly

Public | Public

Protected | Protected

Private | Inaccessible

Access specifier in base class | Access specifier when inherited protectedly

Public | Protected

Protected | Protected

Private | Inaccessible

Access specifier in base class | Access specifier when inherited privately

Public | Private

Protected | Private

Private | Inaccessible

\*/

B b;

b.setData(4);

cout << b.getData() << endl;

}

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

class Vehicle {

public:

Vehicle() { cout << "This is a Vehicle\n"; }

};

// second base class

class FourWheeler {

public:

FourWheeler()

{

cout << "This is a 4 wheeler Vehicle\n";

}

};

// 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;

}

Ambuiguity in Multiple Inheritance-

In multiple inheritances, when one class is derived from two or more base classes then there may be a possibility that the base classes have functions with the same name, and the derived class may not have functions with that name as those of its base classes. If the derived class object needs to access one of the similarly named member functions of the base classes then it results in ambiguity because the compiler gets confused about which base’s class member function should be called.

#include<iostream>

using namespace std;

// Base class A

class A {

public:

void func() {

cout << " I am in class A" << endl;

}

};

// Base class B

class B {

public:

void func() {

cout << " I am in class B" << endl;

}

};

// Derived class C

class C: public A, public B {

};

// Driver Code

int main() {

// Created an object of class C

C obj;

// Calling function func()

obj.func();

return 0;

}

Output:

prog.cpp: In function ‘int main()’:

prog.cpp:43:9: error: request for member ‘func’ is ambiguous

obj.func();

^

prog.cpp:21:10: note: candidates are: void B::func()

void func() {

^

prog.cpp:11:10: note: void A::func()

void func() {

^

In this example, derived class C inherited the two base classes A and B having the same function name func(). When the object of class C is created and called the function func() then the compiler gets confused that which base class member function func() should be called.

Solution to Ambiguity:

To solve this ambiguity scope resolution operator is used denoted by ‘ :: ‘

Syntax:

ObjectName.ClassName::FunctionName();

int main() {

// Created an object of class C

C obj;

// Calling function func() in class A

obj.A::func();

// Calling function func() in class B

obj.B::func();

return 0;

}

Another Solution (using virtual inheritance)

In C++, you can use virtual inheritance to resolve ambiguity in inheritance. Virtual inheritance is a way of specifying that a class should be inherited virtually, meaning that only one instance of the class should be present in the inheritance hierarchy, even if the class is inherited multiple times.

Below is the program to show the concept of ambiguity resolution in multiple inheritances.

#include <iostream>

using namespace std;

class A {

public:

int x;

};

class B : virtual public A {

public:

int y;

};

class C : virtual public A {

public:

int z;

};

class D : public B, public C {

public:

int w;

};

int main() {

D obj;

obj.x = 1; // okay, no ambiguity

obj.y = 2; // okay, no ambiguity

obj.z = 3; // okay, no ambiguity

obj.w = 4; // okay, no ambiguity

return 0;

}

In this example, class A is inherited virtually by classes B and C, so when class D inherits from both B and C, there is no ambiguity in the inheritance of A. As a result, you can access the members of A directly through an instance of D, without any issues.

Note that virtual inheritance can have some performance overhead, as it requires the use of additional pointers to manage the inheritance hierarchy. Therefore, it should only be used when necessary to resolve ambiguity in inheritance.

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

Base class visibility Derived class visibility

Public Private Protected

Private Not Inherited Not Inherited Not Inherited

Protected Protected Private Protected

Public Public Private Protected

class A {

public:

A() {

cout << "In the default constructor of parent A\n";

}

~A() {

cout << "In the destructor of parent A\n";

}

};

class B: public A {

public:

B() {

cout << "In the default constructor of parent B\n";

}

~B() {

cout << "In the destructor of child B\n";

}

};

class C {

private:

int x;

public:

C(int x) {

cout << "Assigned the value of x in Class C. x = " << x << endl;

this->x = x;

}

~C() {

cout << "In the destructor of parent C\n";

}

};

class D: public C {

private:

int y;

public:

/\*

Why this weird syntax?

--> We have created a paratemetrized constructor in out parent class C.

Thus default constructor will not be created by the compiler

--> We aim to initialize values of both x and y using the constructors.

--> So we will have to pass the value to the constructor of the parent class with

which we wish to initialize.

--> Thus this syntax

What is the exact process followed?

1. You create an object of class D, say (2,3)

2. The paramterized constructor of D gets called

3. Before executing the constructor of D, it calls the parametrized constructor of C i.e C(x)

4. One the value of x has been assigned, the value of y gets assigned

IMP Point:

--> Constructor(s) of the derived classes call the constructor(s) of the base class

\*/

D(int x, int y): C(x) {

cout << "Assigned the value of y in Class D. y = " << y << endl;

this->y = y;

}

~D() {

cout << "In the destructor of child D\n";

}

};

int main() {

/\*

Prerequisites:

--> If user does not create any constructor for any class, compiler by default creates a default

and a copy constructor

--> In any class destructor is needed to release the memory allocated to the resources. In a destructor, we can do some

final tasks before an object is removed from the memory.

Order of calling of constructors: From child to parent

Order of execution of constructors: From parent to child

Order of calling of destructors: From child to parent

Order of execution of destructors: From child to parent

\*/

B b;

D d(2, 3);

}

/\*

Object Pointer

--> Pointer which contains address of an object is called object pointer

this Pointer

--> 'this' is a keyword in C++

--> this is a local object pointer in every instance member function and it contains the

address of the caller object

--> this pointer cannot be modified

--> It is used to refer caller object in member function

\*/

#include<bits/stdc++.h>

using namespace std;

int main() {

/\*

SMA: Static Memory Allocation

DMA: Dynamic Memory Allocation (Implemented using new and delete in C++)

In SMA, the amount of memory allocated to the declaration statements(int x, float y, etc) is decided

during Compile time only and cannot be changed. However, the memoery gets allocated when the program

comes in RAM

\*/

int \*x = NULL;

// Requesting memory for a new variable. nothrow signifies that if memory does

// not get assigned it does not throw an error

x = new(nothrow) int;

if (!x) {

cout << "Could not allocate memory\n";

}

else {

\*x = 22;

cout << "Value of x is: " << \*x << endl;

}

float \*y = new(nothrow) float(11.111);

if (!y) {

cout << "Could not allocate memory\n";

}

else {

cout << "Value of y is: " << \*y << endl;

}

// DMA for Arrays

int n = 10;

int \*a = new int[10];

for (int i = 0; i < n; i++) {

a[i] = i + 1;

}

cout << "Array elements are: ";

for (int i = 0; i < n; i++) {

cout << a[i] << " ";

}

cout << endl;

// Deallocating memory for variables

delete x;

delete y;

// Deallocating memory for array

delete []a;

}

class A {

public:

void print() {

cout << "In print function of class A\n";

}

void showData() {

cout << "In showData function of class A\n";

}

};

class B: public A {

public:

void print() { // method overriding

cout << "In print function of class B\n";

}

void showData(int x) { // method hiding

cout << "In showData function of class B\n";

}

};

int main() {

/\*

Method overriding:

--> When you want the same functionalities in your child class but with different functionalities

--> The entire function prototype should be same both in child and parent (No difference allowed)

Method hiding:

--> A function with same name but different arguments in parent and child class

\*/

B b;

/\*

What happens when you do b.print()?

--> When this line is written, during compilation it is the duty of the compiler to bind the call

to the declaration (Early binding)

--> So the compiler searches through the member functions of the caller object and checks for

functions named print().

--> If it finds a function in class B, it stops the search process and binds the call to that function

--> If it does not find, the compiler continues to search it in the parent class and so on

\*/

b.print();

b.A::print();

/\*

Why does the line below cause a CE?

--> During compilation, the compiler starts the process of searching and binding from the

caller class (B)

--> So in this class it has a function with the name showData but with different arguments.

--> If the name matches, the compiler stops searching and that point and the CE is caused

due to mismatch of arguments.

\*/

// b.showData(); Gives CE

b.showData(1);

b.A::showData();

}

virtual fubction:

// Classes A and B are parent & child classes respectively without using virtual functions

class A {

public:

void print() {

cout << "In print function of class A\n";

}

};

class B: public A {

public:

void print() {

cout << "In print function of class B\n";

}

};

// Classes C and D are parent & child classes respectively using virtual functions

class C {

public:

/\*

This virtual keyword tells the compiler that late binding(binding at runtime) should

take place for this function

Working concept behind virtual function link:

https://www.youtube.com/watch?v=Z\_FiER8aAqM&list=PLLYz8uHU480j37APNXBdPz7YzAi4XlQUF&index=45&ab\_channel=C%2B%2BbySaurabhShuklaSir

\*/

virtual void print() {

cout << "In print function of class C\n";

}

};

class D: public C {

public:

void print() {

cout << "In print function of class D\n";

}

};

int main() {

/\*

Base Class Pointer:

--> Base class pointer can point to the objects of any of it's descendant class(es)

--> The converse is not true (Child pointer can't point to parent's objects)

IMP: Binding of function call with function definition happens at Compile Time only

\*/

cout << "Before using virtual function..\n";

A \*p1, a;

B b;

p1 = &b;

a.print();

b.print();

/\*

The main issue with overriding occurs over here.

--> You feel that since p1 contains address of an object of class B, it should bind with the print()

method of class B

--> However, this does not happen in reality. Binding happens at compile time.

--> The function call at this moment does not know the address it will contain. Cause addresses are

assigned at runtime

--> So it checks for the datatype of p1 which is of type A and binds it there

--> Thus, print() function of class A is called

This problem in hand paves way for the need of virtual function

\*/

p1->print();

// Using virtual functions

cout << "\nAfter using virtual function..\n";

C \*p2, c;

D d;

p2 = &d;

c.print();

d.print();

p2->print();

}

#include<bits/stdc++.h>

using namespace std;

// Abstract Class

class Person {

public:

/\*

Virtual keyword is necessary because without this keyword, early binding of

print() function occurs and hence base class function can be called using object

pointers

\*/

virtual void print() = 0;

};

/\*

For the inherited class, either we can override the pure virtual function of base class

or declare the function as pure virtual in child class also.

One of the two has to be done for sure

\*/

class Student: public Person {

public:

void print() {

cout << "In Student Class\n";

}

};

int main() {

/\*

Pure Virtual Function

--> A do nothing function is called Pure Virtual Function

Abstract Class

--> Any class which has one or more pure virtual functions is called Abstract class

--> One cannot create an object of Abstract class (can't instanciate it)

\*/

// Person p; // This is CE

Student s;

s.print();

}

// Function template or Generic template

template<class T>

T maxx(T a, T b) {

if (a > b)

return a;

return b;

}

template<class X>

class ClassName {

// Class elements

};

int main() {

/\*

Template:

--> The keyword template is used to define function and class template

--> It is a way to make any class or function generalized wrt data types

\*/

cout << maxx(2, 3) << endl;

cout << maxx(2.342, 31.121) << endl;

}

int main() {

/\*

--> Exception is any normal behaviour, runtime error

--> Using Exception Handling, a programmer can respond and manage RTEs actively

try, throw and catch

--> Program statements that you want to monitor for exceptions are contained in a try block

--> If any exception occurs within the try block, it is thrown using throw

--> The exception is caught using catch and then processed

IMP Points:

1. try and catch will always be together (Both of them have to exist else no one must exist)

2. throw can be present independently (It will not cause CE but will cause RTE)

\*/

try {

cout << "This will be printed\n";

// throw 3239;

throw ("An error occured");

cout << "This will not be printed\n";

}

catch (char const\* s) {

cout << s << endl;

}

catch (...) { // ... indicates all types of throw statements are allowed

cout << "Non-string exception occured\n";

}

cout << "This will again be printed\n";

// throw 20; // Causes RTE

}

\*Namespace:

Namespace provide the space where we can define or declare identifier i.e. variable, method, classes.

#include<bits/stdc++.h>

using namespace std;

namespace Jinesh {

int x;

void fun();

class Jinesh {

public:

void jinesh();

};

}

void Jinesh::fun() {

cout << "Accessing fun() function from Jinesh Namespace\n";

}

void Jinesh::Jinesh::jinesh() {

cout << "Accessing jinesh() function from Jinesh class inside Jinesh namespace\n";

}

int main() {

/\*

namespace

--> It is a container for identifiers

--> It puts the names of it's members in a distinct space so that they don't conflict

with the names in other global namespace

How to create namespace?

Syntax:

namespace namespaceName{

//Declarations

}

IMP Points:

--> The namespace definition must be done at global scope, or nested inside other namespace

--> You can use an alias name for a namespace

namespace AliasName=namespaceName

--> It is not a class so object cannot be created

--> There can be unnamed namespaces

Syntax:

namespace{

// Declarations

}

--> A namespace definiton can be extended or continued to multiple files, they are not

overwritten or redefined

using directive:

--> using keyword allows a programmer to import an entire namespace into the program

with a global scope

--> It can be used to import namespaces into another namespaces or programs

\*/

Jinesh::x = 2;

Jinesh::fun();

Jinesh::Jinesh j;

j.jinesh();

}

----------------------------------------------------------------------------------------------------------------------

\*Encapsulation:

-->Encapsulation is defined as binding together the data and the functions that manipulate them, in a single unit called class .

-->Consider a company with various divisions like financial, technical, marketing, etc. A division consists of the related records and the employees that manipulate the records. Here, the employees are functions that manipulate the data and are wrapped into a single division.

-->Encapsulation also leads to data abstraction or hiding.

Data hiding:

-->The data can be made only accessible to the functions wrapped in the same unit in an object. These functions provide the interface between the object’s data and the program. This insulation for the data from direct access by the program is called data hiding.

-->Consider the above example 1. The manager wants to access the records of the financial team. They cannot access the records ( data) directly and have to ask the employees ( functions ) for it.

-->This can be achieved using access specifiers, such as ‘private’, and ‘protected’.

-->This ensures that users can’t access internal data without authentication.

-->Getters and setters can be used to access the data or to modify it.

Data abstraction:

-->Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

-->This is usually achieved using ‘abstract’ class concept, and by implementing interfaces.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows −

class Box {

public:

// pure virtual function

virtual double getVolume() = 0;

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called concrete classes.

Abstract Class Example

Consider the following example where parent class provides an interface to the base class to implement a function called getArea() −

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

// pure virtual function providing interface framework.

virtual int getArea() = 0;

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived classes

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

class Triangle: public Shape {

public:

int getArea() {

return (width \* height)/2;

}

};

int main(void) {

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total Rectangle area: " << Rect.getArea() << endl;

Tri.setWidth(5);

Tri.setHeight(7);

// Print the area of the object.

cout << "Total Triangle area: " << Tri.getArea() << endl;

return 0;

}

O/P-->

Total Rectangle area: 35

Total Triangle area: 17

No, you can't use abstract as a keyword because there is no such keyword available in C++.

If you want to declare a C++ class as abstract, you can declare at least one function as a pure virtual function.

But in derived class you must provide a definition otherwise its give compilation error.

Example:

class A

{

public:

virtual void sum () = 0;

};

-----------------------------------------------------------------------------------------------------------------------

Polymorphism:

-->The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

-->A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism.

Types of Polymorphism

Compile-time Polymorphism.

Runtime Polymorphism.

1. Compile-Time Polymorphism consists of two types

A)Function overloading:

-->When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.

B)Operator Overloading

--> operators can be made to perform tasks depending on the operands.

--> For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands.

-->In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.

In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

Now, if the user wants to make the operator “+” to add two class objects, the user has to redefine the meaning of the “+” operator such that it adds two class objects. This is done by using the concept “Operator overloading”. So the main idea behind “Operator overloading” is to use c++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

class Complex {

private:

int real, imag;

public:

Complex(int r = 0, int i = 0) {real = r; imag = i;}

// This is automatically called when '+' is used with

// between two Complex objects

Complex operator + (Complex const &obj) {

Complex res;

res.real = real + obj.real;

res.imag = imag + obj.imag;

return res;

}

void print() { cout << real << " + i" << imag << '\n'; }

};

int main()

{

Complex c1(10, 5), c2(2, 4);

Complex c3 = c1 + c2;

c3.print();

}

line 403--> c1 calls the operartor function with paraamter as c2(object).

2. Runtime Polymorphism:

-->This type of polymorphism is achieved by Function Overriding.

-->Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overriden) in the derived class.

Virtual functions are Dynamic in nature.

They are defined by inserting the keyword “virtual” inside a base class and are always declared with a base class and overridden in a child class

A virtual function is called during Runtime

-->When a function is made virtual, the compiler focuses on the contents of the pointer and ignores the type of the pointer.

-->When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

#include <iostream>

using namespace std;

class A

{

public:

virtual void display()

{

cout << "Base class is invoked" << endl;

}

};

class B : public A

{

public:

void display()

{

cout << "Derived Class is invoked" << endl;

}

};

int main()

{

A \*a; // pointer of base class

B b; // object of derived class

a = &b;

a->display(); // Late Binding occurs

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*O/P--> Derived Class is invoked\*\*\*\*\*\*\*\*\*\*\*\*

int main()

{

A a; // pointer of base class

B b; // object of derived class

// a = &b;

a.display(); // Late Binding occurs

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*O/P--> Base Class is invoked\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main()

{

A a; // pointer of base class

B b; // object of derived class

// a = &b;

a.display(); // Late Binding occurs

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*O/P--> Derived Class is invoked\*\*\*\*\*\*\*\*\*\*\*\*

A virtual function is not used for performing any task. It only serves as a placeholder.

When the function has no definition, such function is known as "do-nothing" function.

The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.

A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.

The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Virtual Destructors

The destructor of derived class is executed before base class.

A virtual destructor is needed when deleting a derived class object using a base class pointer, and the derived class was made in runtime. The derived class destructor is not called at all. ( memory leaking causing data loss )

----------------------------------------------------------------------------------------------------------------------

Inheritance:

-->Inheritance is a mechanism of deriving a new class from an old class so that the new class inherits the properties of the old class and can also have its own new properties.

-->When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class.

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

Super Class: The class whose properties are inherited by a subclass is called Base Class or Superclass.

class Vehicle {

public:

Vehicle()

{

cout << "This is a Vehicle\n";

}

};

// sub class derived from a single base classes

class Car : public Vehicle {

};

// main function

int main()

{

// Creating object of sub class will

// invoke the constructor of base classes

Car obj;

return 0;

}

2.

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

class Vehicle {

public:

Vehicle() { cout << "This is a Vehicle\n"; }

};

// first sub\_class derived from class vehicle

class fourWheeler : public Vehicle {

public:

fourWheeler()

{

cout << "Objects with 4 wheels are vehicles\n";

}

};

// sub class derived from the derived base class fourWheeler

class Car : public fourWheeler {

public:

Car() { cout << "Car has 4 Wheels\n"; }

};

// main function

int main()

{

// Creating object of sub class will

// invoke the constructor of base classes.

Car obj;

return 0;

}

This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

class Vehicle {

public:

Vehicle() { cout << "This is a Vehicle\n"; }

};

// first sub class

class Car : public Vehicle {

};

// second sub class

class Bus : public Vehicle {

};

// main function

int main()

{

// Creating object of sub class will

// invoke the constructor of base class.

Car obj1;

Bus obj2;

return 0;

}

This is a Vehicle

This is a Vehicle

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

class Vehicle {

public:

Vehicle() { cout << "This is a Vehicle\n"; }

};

// base class

class Fare {

public:

Fare() { cout << "Fare of Vehicle\n"; }

};

// first sub class

class Car : public Vehicle {

};

// second sub class

class Bus : public Vehicle, public Fare {

};

// main function

int main()

{

// Creating object of sub class will

// invoke the constructor of base class.

Bus obj2;

return 0;

This is a Vehicle

Fare of Vehicle

#include <iostream>

using namespace std;

class A

{

int a = 4;

int b = 5;

public:

int mul()

{

int c = a\*b;

return c;

}

};

class B : private A

{

public:

void display()

{

int result = mul();

std::cout <<"Multiplication of a and b is : "<<result<< std::endl;

}

};

int main()

{

B b;

b.display();

return 0;

}

Output:

Multiplication of a and b is : 20

------------------------------------------------------------------------------------------------------------------

23. Are class and structure the same? If not, what's the difference between a class and a structure?

No, class and structure are not the same. Though they appear to be similar, they have differences that make them apart. For example, the structure is saved in the stack memory, whereas the class is saved in the heap memory. Also, Data Abstraction cannot be achieved with the help of structure, but with class, Abstraction is majorly used.

25. Are there any limitations of Inheritance?

Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it has some limitations too. Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not correctly implemented, this might lead to unexpected errors or incorrect outputs.

Interface-->The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

35. How is an abstract class different from an interface?

Interface and abstract class both are special types of classes that contain only the methods declaration and not their implementation. But the interface is entirely different from an abstract class. The main difference between the two is that, when an interface is implemented, the subclass must define all its methods and provide its implementation. Whereas when an abstract class is inherited, the subclass does not need to provide the definition of its abstract method, until and unless the subclass is using it.

Also, an abstract class can contain abstract methods as well as non-abstract methods.

37. What is an exception?

An exception can be considered as a special event, which is raised during the execution of a program at runtime, that brings the execution to a halt. The reason for the exception is mainly due to a position in the program, where the user wants to do something for which the program is not specified, like undesirable input.

38. What is meant by exception handling?

No one wants its software to fail or crash. Exceptions are the major reason for software failure. The exceptions can be handled in the program beforehand and prevent the execution from stopping. This is known as exception handling.

So exception handling is the mechanism for identifying the undesirable states that the program can reach and specifying the desirable outcomes of such states.

Try-catch is the most common method used for handling exceptions in the program.

40. Can we run a Java application without implementing the OOPs concept?

No. Java applications are based on Object-oriented programming models or OOPs concept, and hence they cannot be implemented without it.

However, on the other hand, C++ can be implemented without OOPs, as it also supports the C-like structural programming model.

Who developed the first object-oriented programming language?

Alan Kay

Smalltalk was the first language to be developed as a purely object-oriented programming language?

-->C++ is platform dependent and needs to be compiled on every platform. Java is platform-independent. Once it's compiled into bytecode it can be executed on any platform.

C++ was developed by Bjarne Stroustrup

-->C ++ is only compiled and cannot be interpreted.

-->Java can be both compiled and interpreted.

-->Java has support only for object oriented programming models.

-> Python Functions do not have restrictions on the type of the argument and the type of its return value. In C++, the function can accept and return the type of value which is already defined.

->Namespace provide the space where we can define or declare identifier i.e. variable, method, classes.

Diamond problem:

The Diamond Problem is an ambiguity that arises in multiple inheritance when two parent classes inherit from the same grandparent class, and both parent classes are inherited by a single child class.

Without using virtual inheritance, the child class would inherit the properties of the grandparent class twice, leading to ambiguity.

The Diamond Problem is fixed using virtual inheritance, in which the virtual keyword is used when parent classes inherit from a shared grandparent class. By doing so, only one copy of the grandparent class is made, and the object construction of the grandparent class is done by the child class.