1> Union: In C++ unions are exactly similar to those of structure. But the conditions is you can use exactly one fields among several present in it. Helps in better memory management.

Union currency{

Int INR;

Float Pounds;

Char EUR;

};

You can make use of only one data type at once. Size of unions is max size among the all fields.

If I make variable.

Currency eur = 5; //counted as int

Cur = ‘a’ ; //counted as char.

2> Function Overloading: Function with same name and different parameters and return type.

3> Why class when we had structure, cause structure do not support access specifiers.

4> Functions of Class are common for all objects. They get a common memory for all objects.

5> Copy Constructor:

Line::Line(const Line &obj) {

cout << "Copy constructor allocating ptr." << endl;

ptr = new int;

\*ptr = \*obj.ptr; // copy the value

}

Line line2 = line1; // This also calls copy constructor

6> Friend Functions:

Friend can access all private and protected variables of class. Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

friend void printWidth( Box box );

friend class ClassTwo;

7> What if I inherit a class in public scope which has protected variable. Will variables be public or remain protected?

* They remain protected all time you can’t decrease their level. But If you inherit a class in private mode. Then public variables of parent will become private for child. If you inherit as protected then they become protected.

Class Parent {};

Class Child: public Parent {};

8> What will be output of following?

*class* *B1*{

*public:*

    B1(){

        cout<<"Hi B1 !";

    }

};

*class* *B2*{

*public:*

    B2(){

        cout<<"Hi B2 !";

    }

};

*class* *child*: *public* *B1*, *public* *B2*{

};

*int* main(){

*child* c1;

    return 0;

}

Output ->

Hi B1 !Hi B2 !

9> What will be output of following?

*class* *child*{

    child(){

        cout<<"Hi child !";

    }

};

*int* main(){

    fast;

*child* c1;

    return 0;

}

Output -> Error Because access modifier is not given so constructor will be private by default. It will not allow to access constructor

10> What will be output of following?

*class* *B1*{

*public:*

    B1(){

        cout<<"Hi B1 !";

    }

};

*class* *B2*{

*public:*

    B2(){

        cout<<"Hi B2 !";

    }

};

*class* *child*: *public* *B1*, *public* *B2*{

*public:*

    child(){

        cout<<"Hi child !";

    }

};

*int* main(){

    fast;

*child* c1;

    return 0;

}

Output -> Hi B1 !Hi B2 !Hi child !

11> What will be output of following?

*class* *B1*{

*public:*

*void* Func(){

        cout<<"Hi B1 !";

    }

};

*class* *B2*{

*public:*

*void* Func(){

        cout<<"Hi B2 !";

    }

};

*class* *child*: *public* *B1*, *public* *B2*{

};

*int* main(){

    fast;

*child* c1;

    c1.Func();

    return 0;

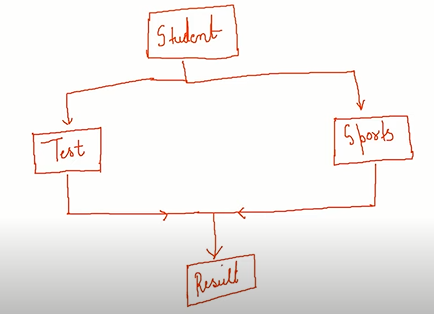
}

Output ->

error: request for member 'Func' is ambiguous

note: To solve ambiguity create a function in child with same name.

12> [Virtual Base Class](https://www.youtube.com/watch?v=eYV-TohBaa0&list=PLu0W_9lII9agpFUAlPFe_VNSlXW5uE0YL&index=45):



What if Student has variable like ID. ID will get inherited in both classes Test and Sports class. And We derived a Result from it it will create ambiguity about which ID it takes of Test or of Sports. For That we can inherit class virtually. Refer Virtual Base Class txt file in folder for program representation.

13 > Guess the Output:

*class* *Student*{

*public:*

*int* ID = 1;

};

*class* *Test*: *public* *Student*{

};

*class* *Sports*: *public* *Student*{

};

*class* *Results*: *public* *Test*, *public* *Sports*{

*public:*

*void* show(){

        cout<<"ID =  "<<ID;

    }

};

*int* main(){

    fast;

*Results* r1;

    r1.show();

    return 0;

}

Output ->

error: reference to 'ID' is ambiguous

To fix it use virtual public or public virtual while inheriting as follows:

*class* *Student*{

*public:*

*int* ID = 1;

};

*class* *Test*: *virtual* *public* *Student*{

};

*class* *Sports*: *public* *virtual* *Student*{

};

*class* *Results*: *public* *Test*, *public* *Sports*{

*public:*

*void* show(){

        cout<<"ID =  "<<ID;

    }

};

*int* main(){

    fast;

*Results* r1;

    r1.show();

    return 0;

}

14> Polymorphism:

1. Compile Time Polymorphism (Static Binding, Early Binding): Object bound to function during compile time itself.

Ex. -> Function, Operator Overloading.

1. Dynamic (Dynamic Binding, Late Binding): Compiler do not which one to run decides at run time.

Ex. -> Virtual Functions, Function Overriding.

15 > Guess the output:

*class* *Base*{

*public:*

*void* show(){

        cout<<"Base ";

    }

};

*class* *Derv*: *public* *Base*{

*public:*

*void* show(){

        cout<<"Derv ";

    }

};

*int* main (){

*Base* \*b1;

*Base* objBase;

*Derv* objDerv;

    b1 = &objDerv;

    b1->show();

    return 0;

}

Output -> “Base” as Pointer type is of base class it will call base class function.

But if you declare base class method as virtual function. Then derived class function will be called. Virtual function overrides default behavior, instead of calling base it call derive function.

16> To call constructor of base from derived

class CWHVideo: public CWH

{

int videoLength;

public:

CWHVideo (char \*s, float r, int vl): CWH(s, r) {

// This line will call Base class constructor 1st and then derived class.

}

17> Rules for virtual functions

1. They cannot be static

2. They are accessed by object pointers

3. Virtual functions can be a friend of another class

4. A virtual function in base class might not be used i.e., they can be empty.

18> An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* ()=0; in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

class AB {

public:

virtual void func() = 0;

//you can not define this func here even if you add {} will throw error

};

Difference in pure virtual and virtual is that if pure virtual function must be overridden in derived class else it will throw an error. Virtual if not overridden it automatically calls function which is present in parent class.

18> Templates:

*template* <*class* *T*>

*class* *DataType*{

*public:*

*T* val;

    DataType(*T* *value*){

        val = *value*;

        cout<<"\nValue is = "<<val;

    }

};

*int* main (){

*DataType* d1(5);

*DataType* d2(5.5);

*DataType* d3("Vinayak");

    return 0;

}

19 > What is initializer list why we need it?

Firstly we need initializer list to initialize const and & reference variables inside our class.

Second reason is below mentioned example.

class A

{

    private:

    int iVal;

    public:

    A(int iNum) : iVal{iNum}

    {

    }

};

class B

{

    private:

    A objA; // For this we need initializer list

    public:

    B() : objA{100}

    {

    }

};

int main()

{

    cout<<"EXIT";

    return 0;

}

If class A has a parametrized constructor it’s default constructor will be vanished automatically.

Hence in class B statement A objA; will not be able to create object. It can be done only using initializer list there is no other option in such cases.

20> Virtual functions:

{IMP: Pure virtual functions can also have a body}

1. VTables are created the moment we declare any function as a virtual. These Vtables are static arrays i.e., they are subjected to a particular class not object it has list of functions.
2. Also if parent class has any virtual function separate VTable for derived class will also get created.
3. Second thing that compiler does is that moment we declare virtual function a compiler automatically adds Vptr inside a class. This Vptr points to VTable of that class.
4. For any derived class if parent has virtual function parent Vptr will get inherited in derived class.
5. Vptr is initialized inside a constructor this code automatically done by compiler.
6. Unlike VTable Vptr are not subject to class they are assigned according to object.

Ex.

Employee \*pObjEmp = new Engineer();

Vptr is initialized according to Engineer as Vptr is created per object and initialized inside a constructor.

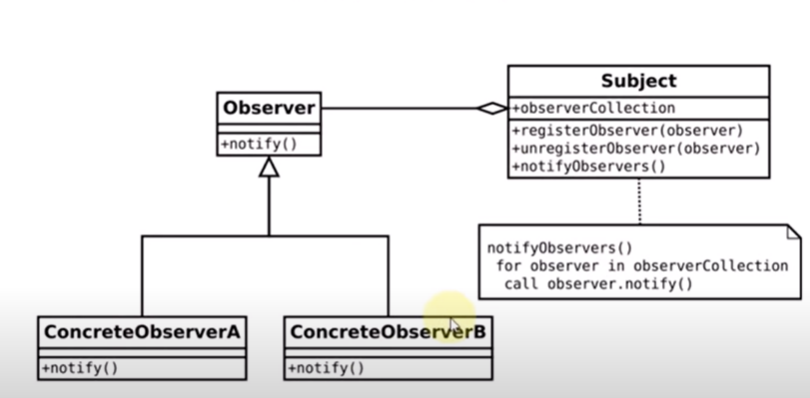
1. Virtual functions can’t be static as they get called as per object.

21> Virtual Constructor/ Destructor:

1. Virtual constructor is not possible because Vptr to point functions gets initialized inside a constructor itself. Compiler itself will give an error in this case.
2. Whenever we have virtual functions make sure we have virtual ~destructor() in our base class otherwise it will create a memory leak.

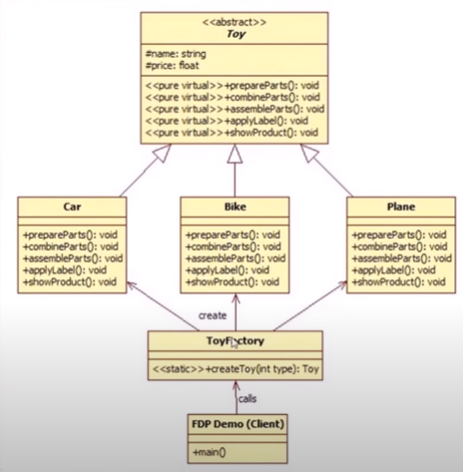
22> Observer Design Patterns:

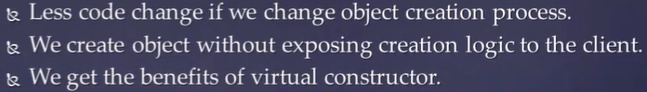
1. Observer design pattern is used to notify other object when state of particular object changes. Subject class has methods like registerObserver(), unregisterObserver(), and notifyObserver().



23> Factory Design Pattern:

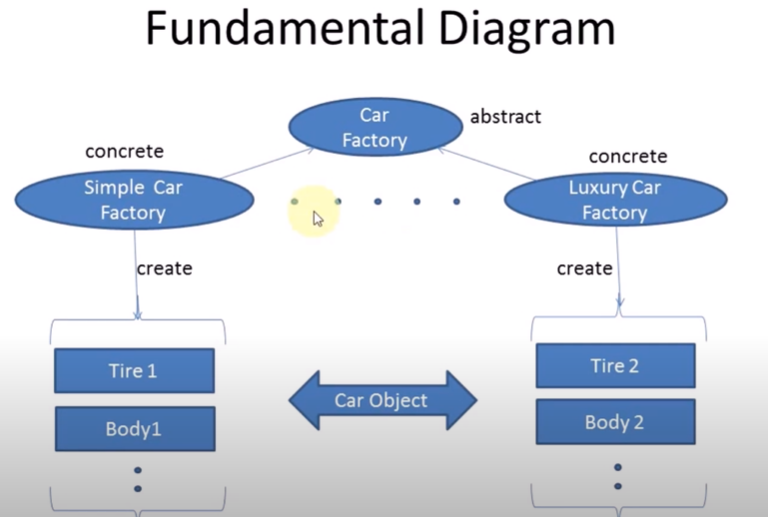
1. If we want to create same types of object then we use factory method to build these kind of objects. We can pass parameters to factory and accordingly we will get our object ready.





1. With help of factory we can get our object at a runtime.

24> Abstract Factory Design Pattern:



25> Singleton Design Pattern:

1. When we want only one object of class and global access to that class.

For Ex. For common and shared resource and for database tables and files.

Or a logger class where you want to log errors and output.

Or to access thread pool or App settings/ Game Settings class. Etc.

1. When we create any class member as a static we have to declare it outside class as mentioned below so that it will get it’s memory otherwise compiler will throw an error. Saying undefined reference.

};

Singleton\* Singleton::pObj;

std::mutex Singleton::singleMutex;

26> const Member functions:

Member functions can be declared const to indicate that they do not modify the object. Such functions can only call other const member functions and cannot modify any member variables.

class MyClass {

public:

    int value;

    MyClass(int val) : value(val) {}

    void showValue() const {

        // value = 20;  // This will cause a compilation error

        std::cout << "Const member function: " << value << std::endl;

    }

};

int main() {

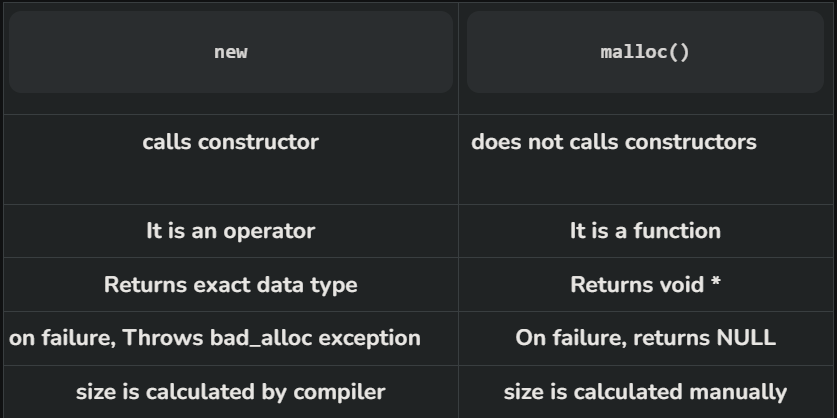
    MyClass obj(42);

    obj.showValue();  // Output: Const member function: 42

    return 0;

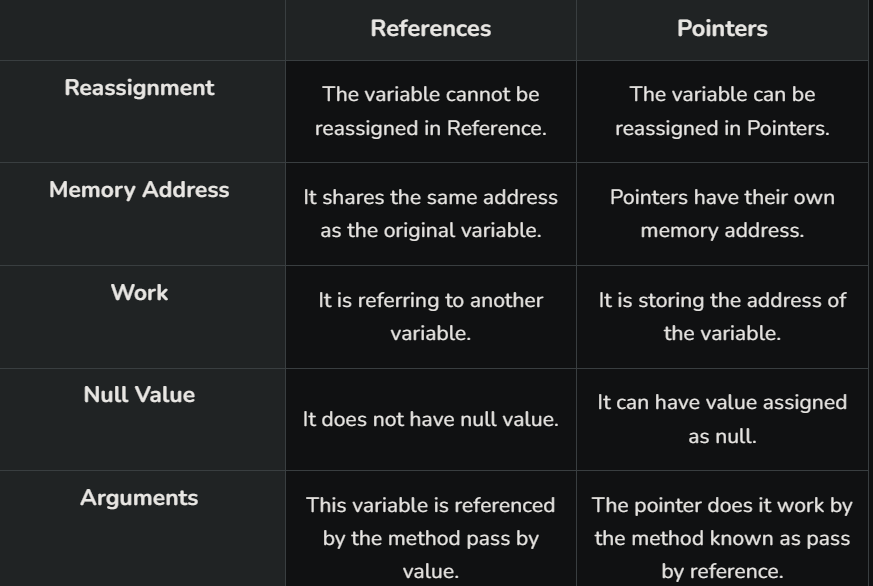
}

27> Malloc vs new:



28> Pointers and Referances:

Pointer can have n number of indirections i.e., int \*\*\*\*p;



29> How do you prevent a class from being inherited in C++?

Using private constructor.

30> Dimond Problem:

The diamond problem is a complication that arises in multiple inheritance scenarios in object-oriented programming, particularly in languages like C++ that support it. It occurs when a class inherits from two or more classes that have a common ancestor. This results in ambiguity regarding which inherited method or attribute should be used when accessed through the derived class.

Consider the following scenario:

```plaintext

A

/ \

B C

\ /

D

```

In this scenario, classes `B` and `C` both inherit from class `A`, and class `D` inherits from both `B` and `C`. Now, if class `A` has a method `foo()`, and both `B` and `C` override `foo()` with different implementations, which implementation should `D` inherit? This ambiguity is the diamond problem.

### C++ Resolves the Diamond Problem Using Virtual Inheritance:

C++ resolves the diamond problem through the use of virtual inheritance. When a class is virtually inherited, a single copy of its base class is shared among all the derived classes that inherit from it. This ensures that there is no duplication of inherited members, resolving the ambiguity.

```cpp

class A {

public:

void foo() { cout << "A::foo()" << endl; }

};

class B : virtual public A {

public:

void foo() override { cout << "B::foo()" << endl; }

};

class C : virtual public A {

public:

void foo() override { cout << "C::foo()" << endl; }

};

class D : public B, public C {

};

int main() {

D d;

d.foo(); // Output: C::foo() - No ambiguity due to virtual inheritance

return 0;

}

```

In this example, both classes `B` and `C` inherit virtually from class `A`. As a result, when class `D` inherits from both `B` and `C`, it has only one copy of the shared base class `A`. Thus, there is no ambiguity regarding which implementation of `foo()` to use when calling it through `D`.

By using virtual inheritance, C++ ensures that the diamond problem is resolved, and the inheritance hierarchy is properly structured, avoiding duplication and ambiguity in inherited members.