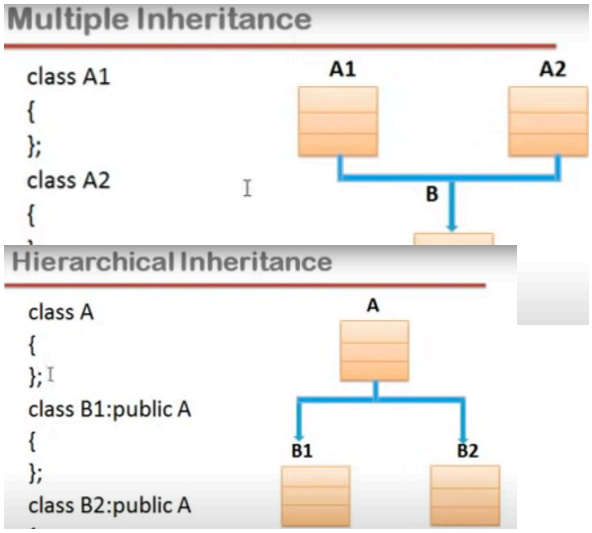
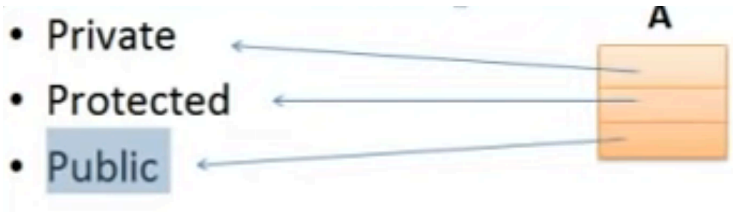
INHERITANCE

Types of Inheritance

1. Single A->B
2. Multilevel A->B->C
3. Multiple A,B ->C
4. Hierarchical A-> B,C
5. Hybrid (any combo of above)

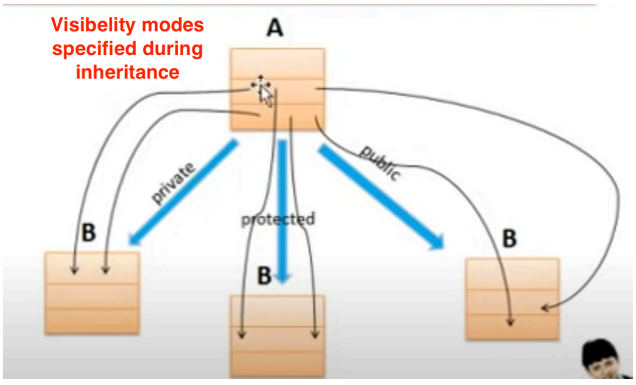


ACCESS SPECIFIERS:



- Private members are accessible only to the parent class , they will neither be accessible when you create an object nor when you derive a class from it
- Protected members CANNOT be accessed when an object is created but is accessible to a derived class
- Public members are always accessible , both when you create and object or derive a class from it - they are the only members accessible when an object is created

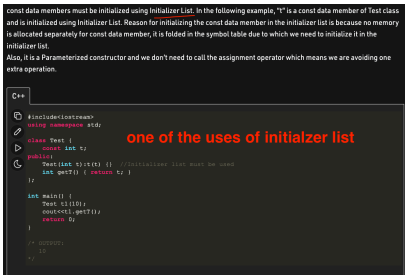
VISIBELITY MODES:(inheritance)



- THESE VISIBILITY MODES COME INTO USE WHEN WE CREATE A OBJECT OF DERIVED CLASS
- regardless of which mode you use child can access protected and public variables of parent class but not private member of parent class
- When inherited as :private , protected and public members of parent class become private in derived class
- When inherited as :protected , protected and public members of parent class become protected in derived class
- When inherited as :public , protected and public members of parent class become protected and public in derived class

ASSOCIATION IN CLASSES

1. Aggregation
2. Composition
3. Inheritance.




```
class Array{
private:
 int a[10]; "stack is NOT an array"
public:
 void insert(int index, int value)
 { a[index]=value;}
};
class STACK: public Array
{
 int top;
public:
 void push(int value)
 {
  insert(top,value);
 }
};
void main()
{
 STACK s1;
 s1.push(34);
 s1.insert(2,56);
}
```

private

WRONG

- ("Is -a " relationship) always uses public inheritance
- We use private/protected inheritance when we want to curb the use of protected/public members we used to create our class by anyone who creates our class !

CONSTRUCTORS AND DESTRUCTORS IN INHERITANCE

- child class implicity calls constructor of parent class
- If parent class constructor needs arguments then child constructor and parent contractor call needs to be called via Initializer List
- first child destructor is executed then parent destructor is called - exactly opposite of constructor order

```
class A
{
 public:
  A()
  { }
};
class B: public A
{
 public:
  B():A()
  { }
};
void main()
{
 B obj;
}
```

```
class A
{
  int a;
 public:
  A(int k)
  { a=k;}
};
class B: public A
{
  int b;
 public:
  B(int x,int y):A(x)
  {b=y; }
};
void main()
{
 B obj(2,3);
}
```

NOTE: Initializer List

https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/

STAITC KEYWORD

STATIC LOCAL VARIABLE

- these variables persist until program ends
- They are initialised only once , not made again and again
- They have a default value of 0

STATIC MEMBER VARIABLE

- *Must be defined outside the class !*
- As we create more object each has instance variables but NOT static member variables - ye class ka variable hai object ka nail

STATIC MEMBER FUNCTIONS

- can only access static variables

A static variable is a variable that is declared using the keyword static. The space for the static variable is allocated only one time and this is used for the entirety of the program.

Once this variable is declared, it exists till the program executes. So, the lifetime of a static variable is the lifetime of the program.

A program that demonstrates a static variable is given as follows.

Example

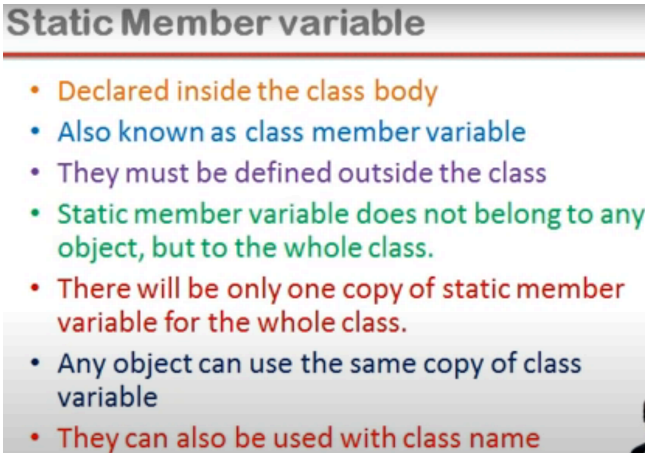
Live Demo

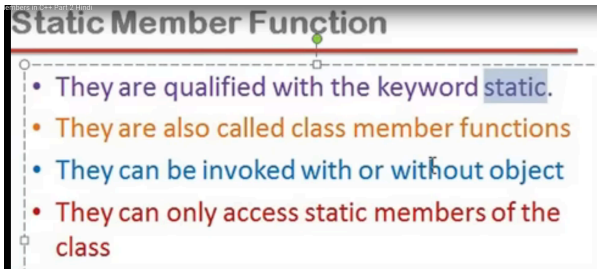
```
#include <iostream>
using namespace std;
void func() {
   static int num = 1;
   cout <<"Value of num: "<< num <<"
";
   num++;
}
int main() {
   func();
   func();
   func();
   return 0;
}
```

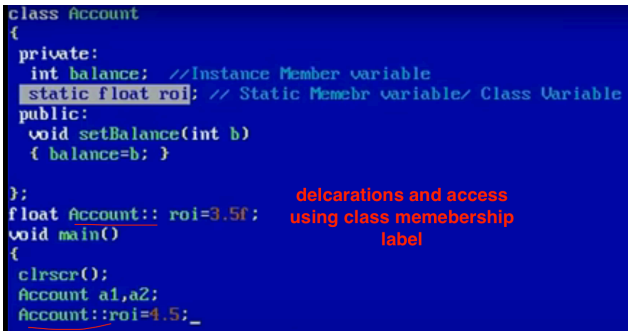
Output

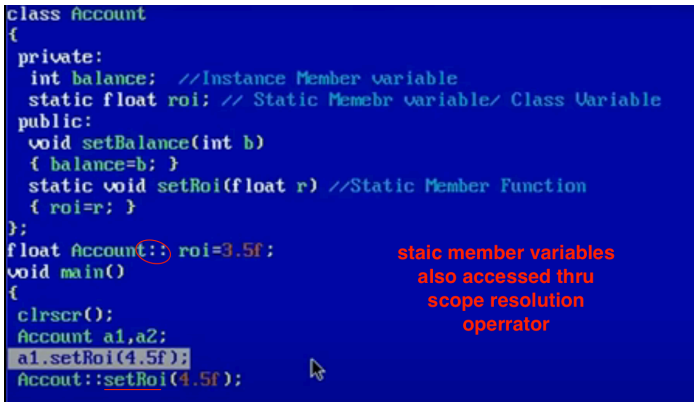
The output of the above program is as follows.

```
Value of num: 1
Value of num: 2
Value of num: 3
```





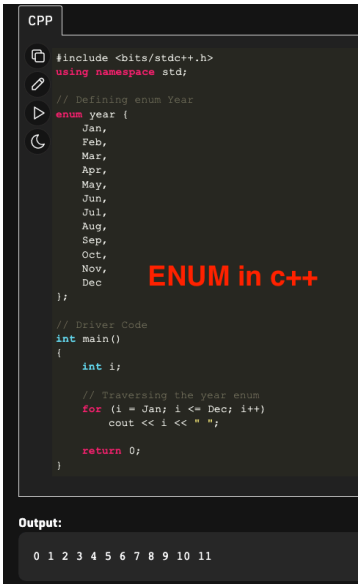




NOTE: ENUM

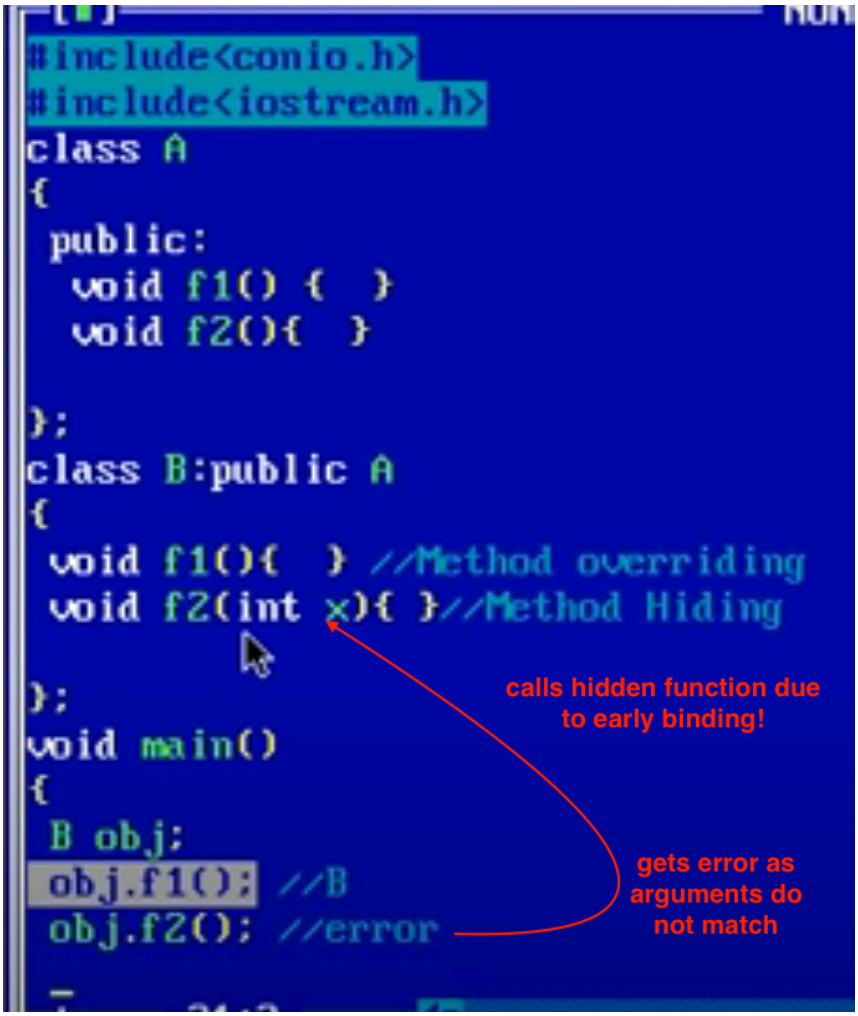
SCOPE RESOLUTION IS ALSO USED HERE

Ie cout<<year::jan<<endll;//0



METHOD OVERLOADING/OVERRIDING/HIDING

- *You* **overload** *a method when the function has same name but different parameters, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.*
- *Allows a derived class to provide its own implementation of a* **virtual function inherited from a base class.** *The derived class overrides the behavior of the base class function, providing a specialized implementation. Method overriding enables polymorphism and dynamic dispatch, where the appropriate function implementation is selected based on the runtime type of the object.*
- *You* **hide** *a method when you define a method in derived class of same name as the non virtual function in base class it may even have different arguments. (But why can't the parents function still be called using appropriate arguments then ?? - the answer is because of early binding , now parents function will NEVER be called !!!!)*



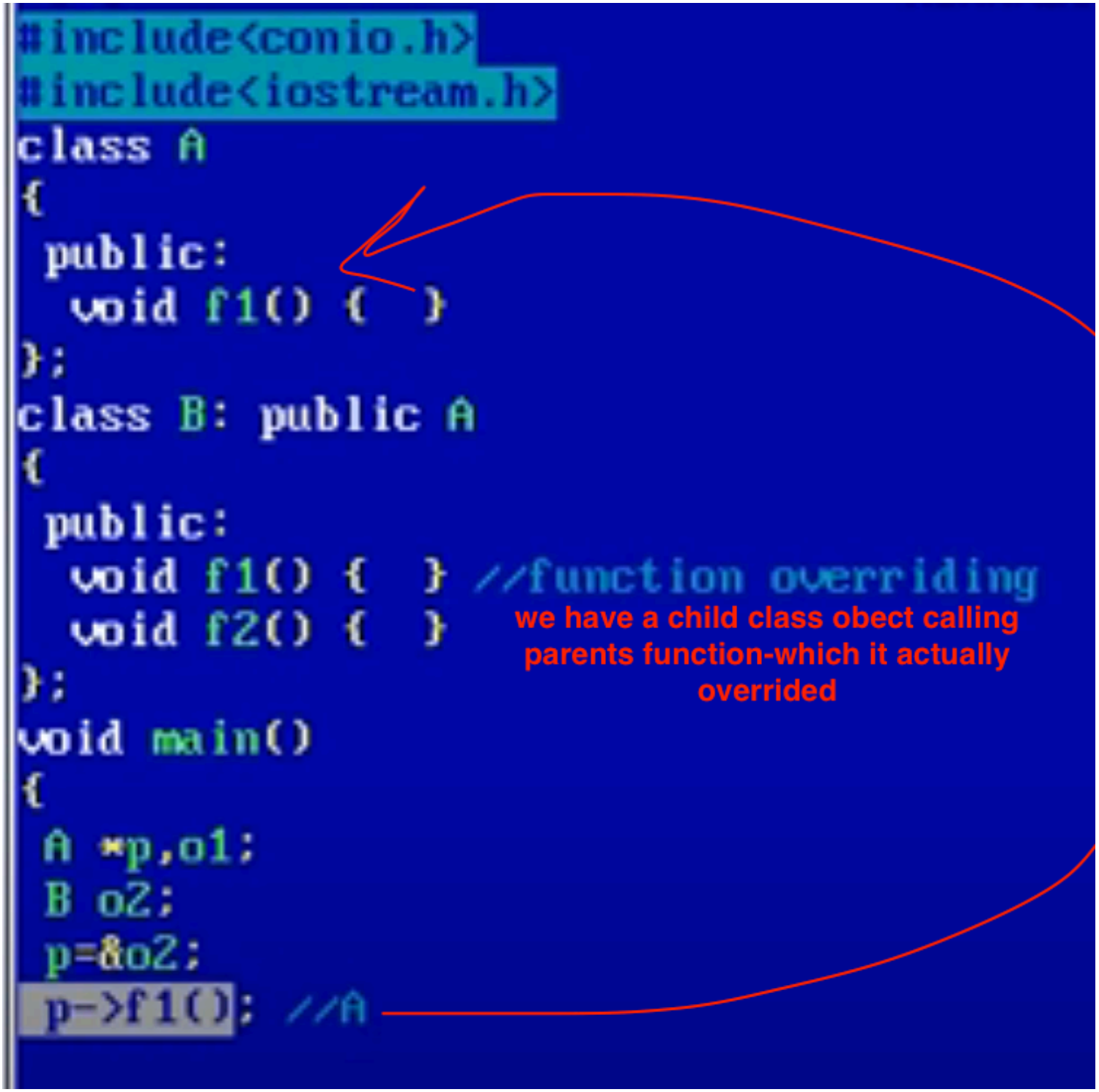
-EARLY BINDING

-When we access a function using an object , its compiler duty to fix which function will run

-Early binding takes the OBJECT TYPE as basis , so it binds calls with the respective classes functions

VIRTUAL FUNCTIONS

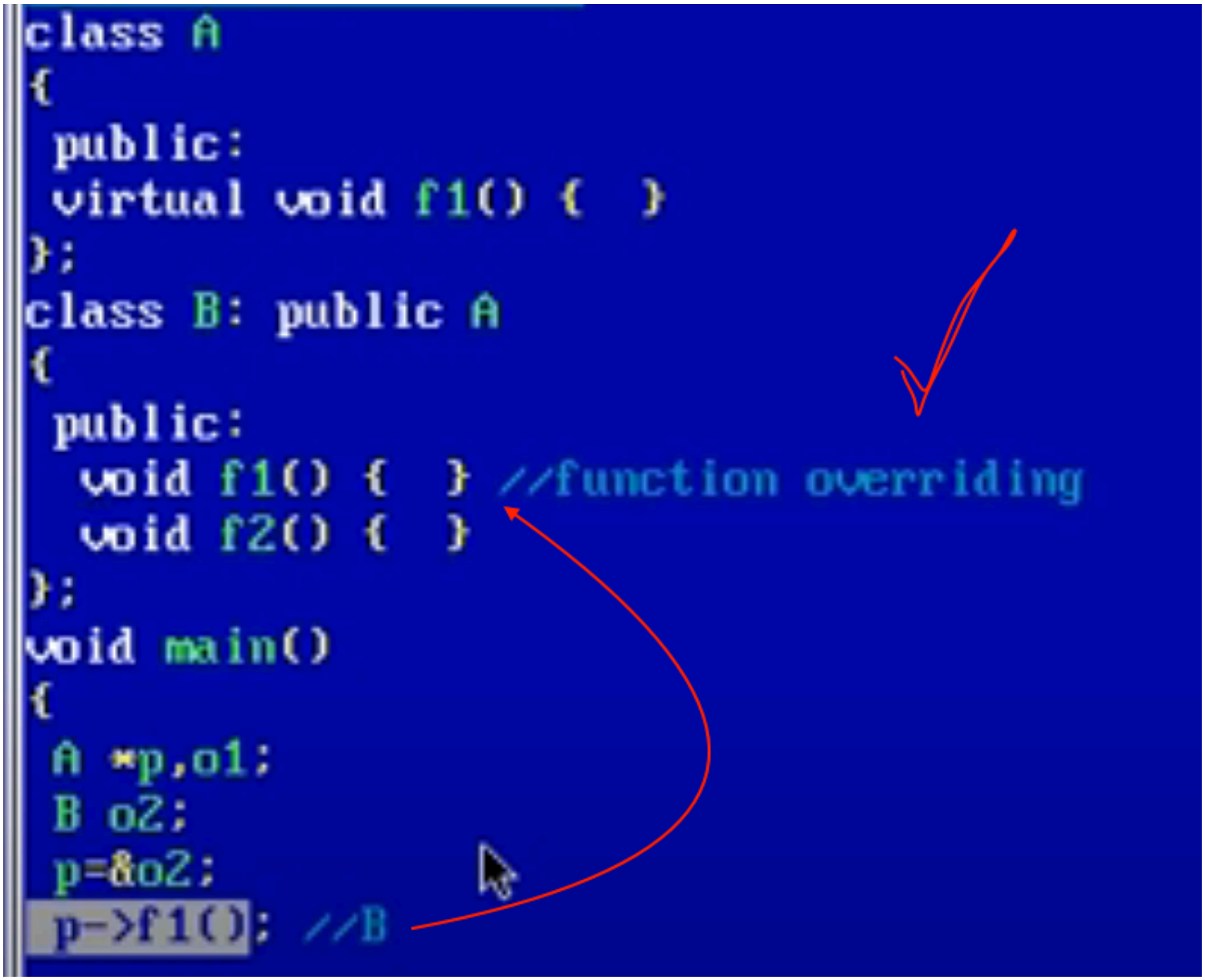
- *Base class pointer can point to object of child class . But child class pointer cannot point to object of parent class*
- *Now when a base class pointer, actually pointing to a child class object , calls a function which is overridden by child, the parent class function is called -due to early binding (compiler takes type of caller/pointer as base to bind)*



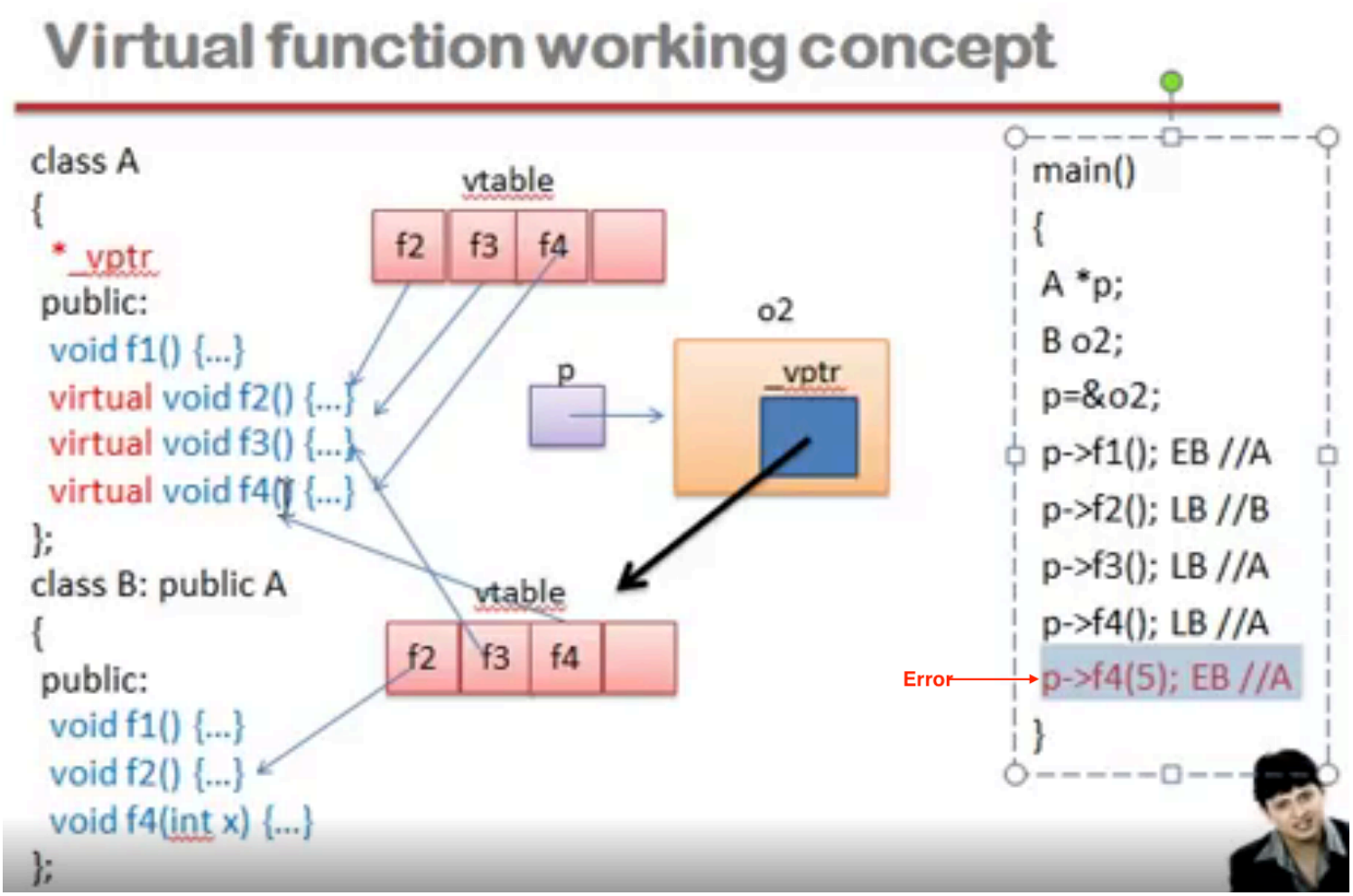
-

-

- *The solution is to bind at runtime-LATE BINDING*
- *Any function marked virtual will be subjected to late binding*
- **Now instead of pointer type pointer content is used as basis to decide which function is to be called**



- *Every function overriding a virtual function , is also by default a virtual function*
- *For every virtual class ,compiler by default adds a variable member , called VPTR pointer and a static array of function pointers -VTABLE, this contains address of all virtual functions on that class*

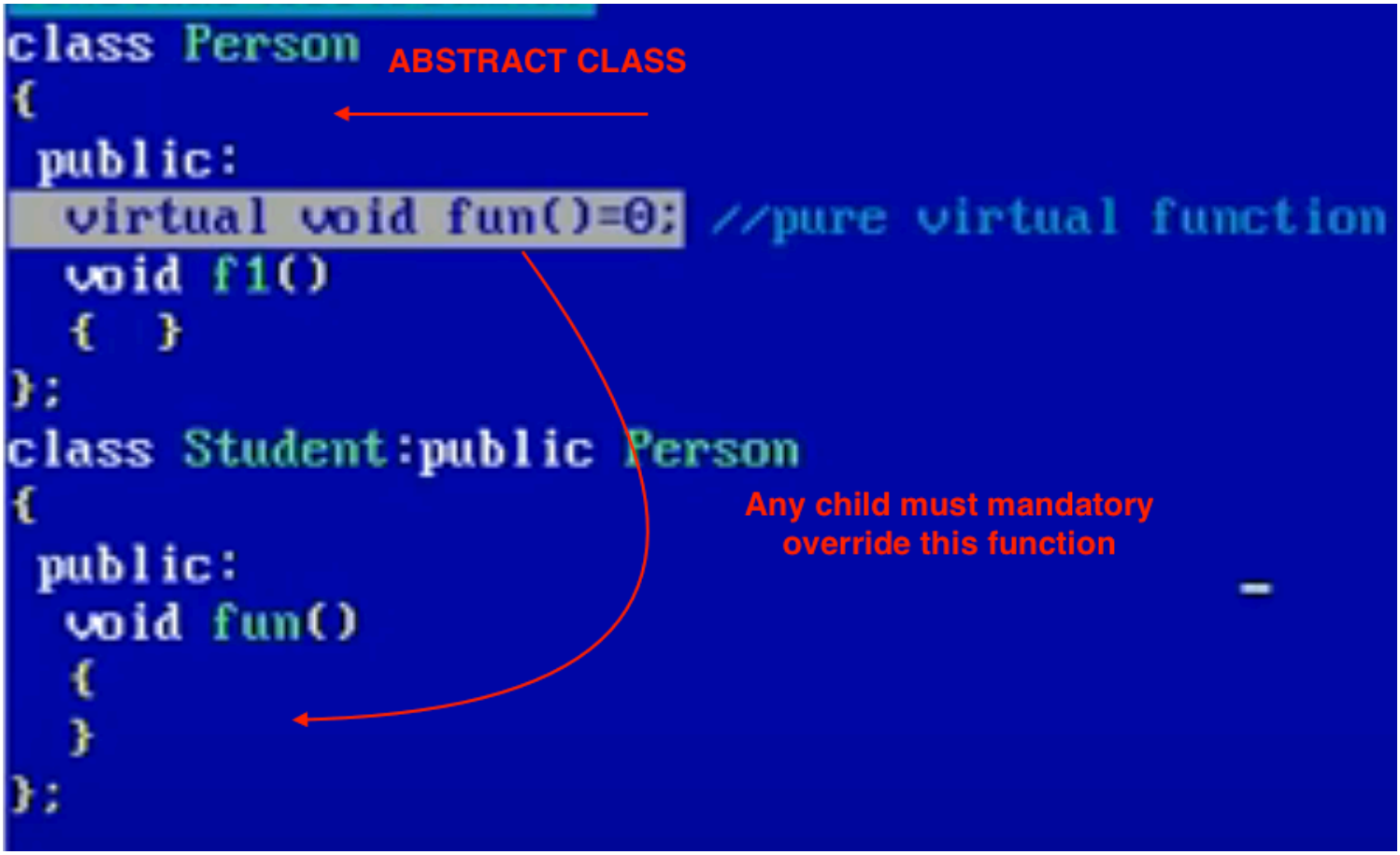


- *Above f4 in child class is not a virtual function ! - as it is not overriding but hiding !*
- *(above) early binding will see type of p and not content , and bind call to parents A , since there is no argument in f4 present in base class , p->f4(5) will throw error*

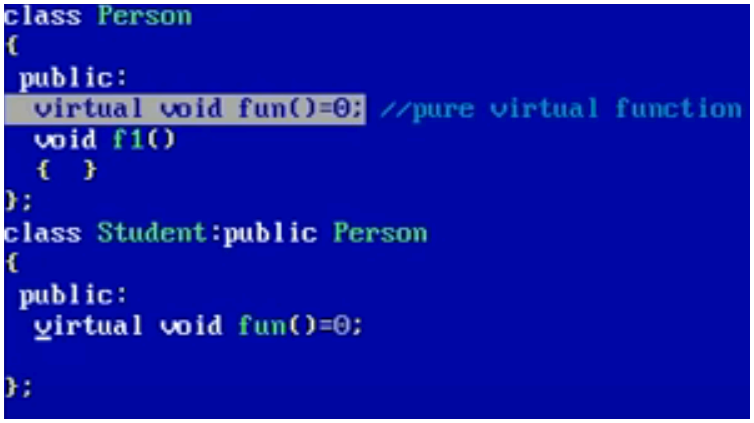
ABSTRACT CLASSES IN C++

1. Object cannot be created for class containing do -nothing functions- create a child class to access some of its normal functions
2. In the child class is do-nothin function must be overridden

- *A* **pure virtual function is a function which has no definition**
- *A pure virtual function us defined by a =0 at the end*
- *As we can see calling a pure virtual function would be useless , so the you should not be able to create an object of the class containing a virtual function*
- *You should not even be able to call this pure virtual function of you derive a child class and try to call from that , so you will have to mandatorily override that pure virtual function in the child class*
- *You should not be even be able to call this function when base class pointer (pointing to child class object)calls this function - that is the reason why its made visual*
- *Any class containing at least one Pure virtual function -is an abstract class ,you cannot make an object of abstract class*



- *Only way for child to prevent overriding is to declare PVC of parent as PVC again with itself - MAKING ITSLEF ALSO AN ABSTRACT CLASS*
-



```
class Person
{
 public:
  virtual void fun()=0; //pure virtual function
  void f1()
  {  }
};
class Student:public Person
{
 public:
  virtual void fun()=0;

};
```

INTERFACE IN C++

I assume that with **interface** you mean a C++ class with only *pure virtual* methods (i.e. without any code), instead with **abstract class** you mean a C++ class with virtual methods that can be overridden, and some code, but *at least one pure virtual method* that makes the class not instantiable. e.g.:

```
class MyInterface
{
public:
  // Empty virtual destructor for proper cleanup
  virtual ~MyInterface() {}

  virtual void Method1() = 0;
  virtual void Method2() = 0;
};


class MyAbstractClass
{
public:
  virtual ~MyAbstractClass();

  virtual void Method1();
  virtual void Method2();
  void Method3();

  virtual void Method4() = 0; // make MyAbstractClass not instantiable
};
```

1. Interface class does not have any method implementation. It only has method declarations and the class that implements an interface implement the methods.
2. Interface does not have defined variables. It does exist in java but then the variables are stated as final and static.
3. The class which implements an interface must implement all the methods of the interface.
4. Abstract class can have variable declaration and method implementation/declarations. Moreover one can inherit the abstract class without implementing the abstract methods.
5. An abstract class cannot be instantiated but rather inherited by another class. Instantiating and abstract class will give compilation error.

Override keyword

The `override` keyword serves two purposes:

1. It shows the reader of the code that "this is a virtual method, that is overriding a virtual method of the base class."
2. The compiler also knows that it's an override, so it can "check" that you are not altering/adding new methods that you think are overrides.

To explain the latter:

```
class base
{
  public:
    virtual int foo(float x) = 0;
};


class derived: public base
{
   public:
     int foo(float x) override { ... } // OK
};

class derived2: public base
{
   public:
     int foo(int x) override { ... } // ERROR
};
```

In `derived2` the compiler will issue an error for "changing the type". Without `override`, at most the compiler would give a warning for "you are hiding virtual method by same name".

Virtual Function in C++

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

Pure Virtual Functions in C++

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation, we only declare it. A pure virtual function is declared by assigning 0 in the declaration.

Similarities between virtual function and pure virtual function

1. These are the concepts of Run-time polymorphism.
2. Prototype i.e. Declaration of both the functions remains the same throughout the program.
3. These functions can't be global or static.

Difference between virtual function and pure virtual function in C++

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
`virtual<func_type><func_name>()`
`{`
`  // code`
`}`
	Syntax:
	`virtual<func_type><func_name>()`
	`  = 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

NOTE: enum class in c++
https://www.geeksforgeeks.org/enum-classes-in-c-and-their-advantage-over-enum-datatype/
https://stackoverflow.com/questions/18335861/why-is-enum-class-preferred-over-plain-enum

NOTE: explicit constructor in c++
https://stackoverflow.com/questions/121162/what-does-the-explicit-keyword-mean

Static Variables:

1. Local Static Variables: A local static variable is a variable declared within a function with the "static" keyword. It retains its value between function calls and is only initialized once, unlike a regular local variable which is reinitialized every time the function is called.

Example:

c Copy code

```c
#include <iostream>

void func()
{
    static int count = 0;
    count++;
    std::cout << count << std::endl;
}

int main()
{
    func();
    func();
    func();
    return 0;
}
```

2. Static Member Variables: A static member variable is a variable that is declared within a class with the "static" keyword. It is shared among all objects of that class, rather than being unique to each object. It is only initialized once, when the program starts.

Example:

c Copy code

```c
#include <iostream>

class Counter
{
public:
    static int count;
};

int Counter::count = 0;

int main()
{
    Counter a, b, c;
    a.count++;
    b.count++;
    c.count++;
    std::cout << Counter::count << std::endl;
    return 0;
}
```

Static Member Functions:

A static member function is a function declared within a class with the "static" keyword. It can only access static member variables and does not have access to `this` pointer as it does not belong to any particular object. It can be called using the class name and the scope resolution operator (::).

Example:

c Copy code

```c
#include <iostream>

class Counter
{
public:
    static int count;
    static void increment() { count++; }
};

int Counter::count = 0;

int main()
{
    Counter::increment();
    Counter::increment();
    Counter::increment();
    std::cout << Counter::count << std::endl;
    return 0;
}
```

Regenerate response

- local Static variables are used to maintain state info between function calls, also helpful for memory management
- Class Static variables are used to maintain data that is shared between all objects of the class
- Member static functions are used to perform operations that are independent of any particular object
-

Resource Allocation: Local static variables can be used to allocate resources such as file handles, sockets, or database connections that need to persist throughout the lifetime of the program. By using local static variables, you can ensure that the resource is only allocated once, even if the function is called multiple times.

Global State Management: Static member variables allow you to store data that is shared between all objects of a class. This can be useful for managing global state information, such as the number of instances of a class that have been created or the number of times a class method has been called.

API's (Application Programming Interfaces) typically do not violate the Dependency Inversion Principle (DIP) for the following reasons:

1. Abstraction: API's provide an abstraction layer between the client and the implementation of a service. This abstraction layer allows the client to interact with the service without having to know the details of its implementation, thereby making the client code decoupled from the implementation details of the service.
2. Contract: API's define a contract that specifies the expected inputs and outputs for a service. This contract defines the dependencies between the client and the service, but it does not prescribe how the service should be implemented.
3. Reversed Dependencies: The client depends on the API, and not on the implementation of the service. This reversed direction of dependencies makes it easier to change the implementation of the service without affecting the client code, since the client is only dependent on the API and not on the implementation details of the service.

In general, API's can be considered as adhering to the Dependency Inversion Principle as they provide an abstraction layer and a contract that separates the client code from the implementation details of a service, thereby reducing the dependencies between the two and making the system more maintainable and scalable.