Sunbeam Infotech

Exploring new ideas, Reaching new heights!

Virtual function

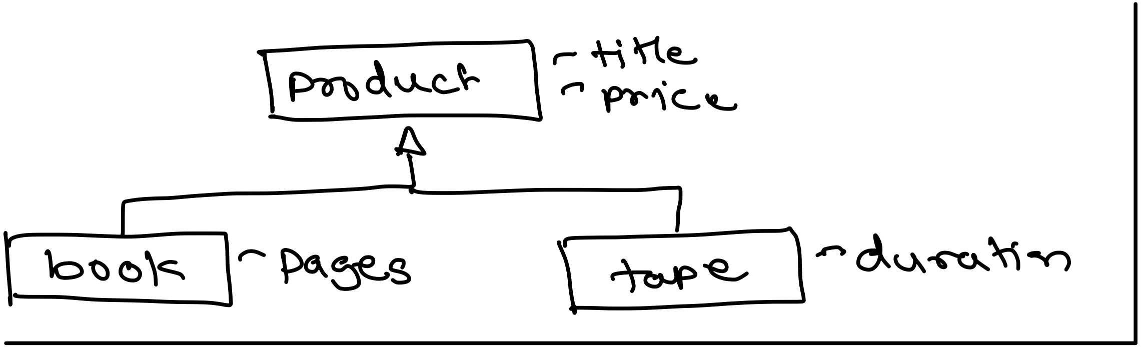
- Virtual function is declared with virtual keyword in C++. → declaration of member function.
- If behaviour of any function is expected to be different than base class function, then that function is declared as virtual in base class and redefined in derived class with same signature.
- Redefining method in derived class with same signature is called as "function overriding".
- Virtual function is always called depending on type of object (not on type of caller).
  → object
  → pointer
  → reference

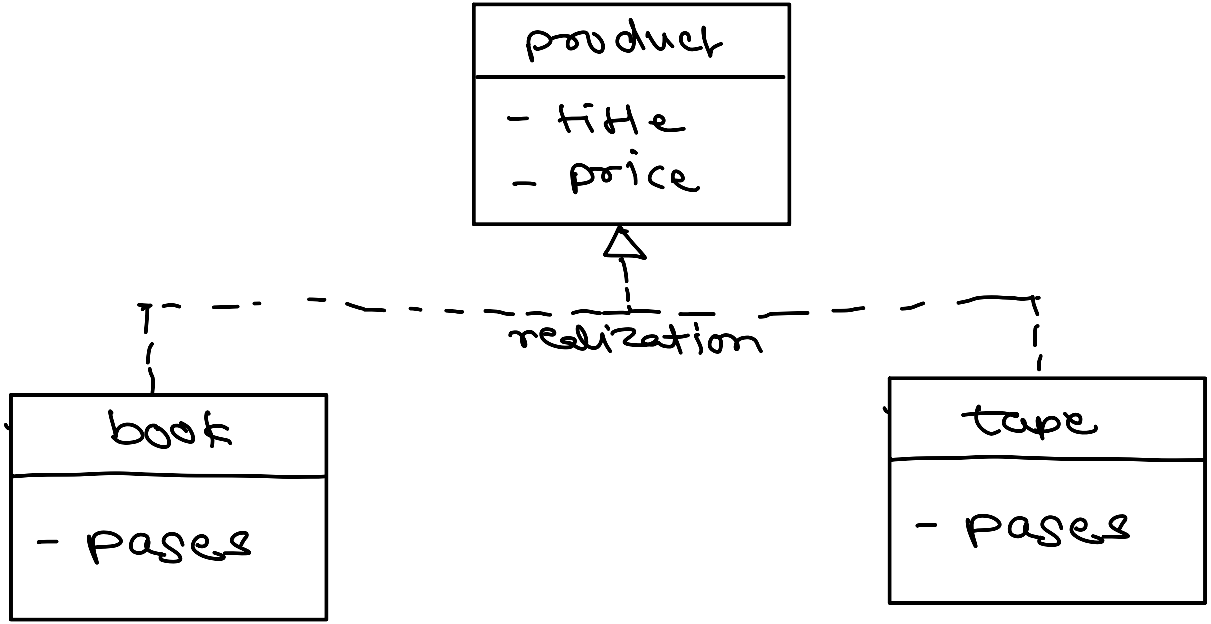
Method overloading vs Method overriding

runtime polymorphism

- Methods with same name and different arguments. → in same scope.
- Based on arguments each method will be given different name internally by the compiler, called as "Name mangling". → false polymorphism
- Method to be called is decided by the compiler, depending on arguments. It is also referred as "Early binding".
- Return type of method is not considered (because collecting retturn value is not compulsory).
- Constructors can be overloaded, but destructors cannot be overloaded.

- Method with same prototype redefined in derived class.
- Internally use virtual function table for the class and virtual table pointer for the object, to invoke appropriate method at runtime. → true polymorphism.
- Method to be called is decided at runtime, depending on type of object. It is referred as "Late binding".
- Return type of method in derived class must be same or sub-type of return type as in base class.
- Constructor cannot be virtual, but destructor can be virtual (for special cases).


product
- title
- price
book
- pages
tape
- duration


product
- title
- price
realization
book
- pages
tape
- pages

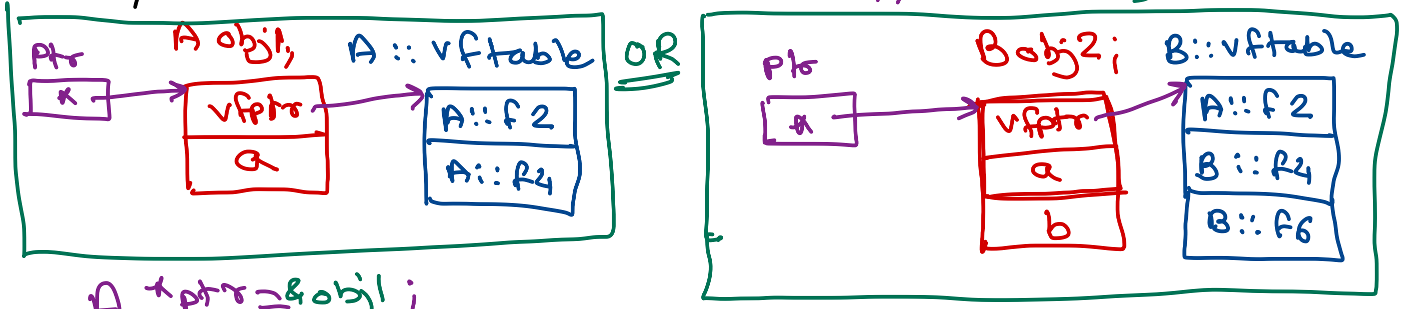
Virtual function table & Virtual function table pointer

- If class contains at least one virtual function, vtable is created for the class.
- It contains addresses of virtual functions in that class.
- Vtable of derived class is made from vtable of base class and entries of overridden methods are updated.
- If class contains at least one virtual function, vptr is added as first data member of the class.
- The vptr always points to vtable of that class. This initialization is done by constructor of the class.
- Late binding calls are resolved using vptr & vtable at runtime.

```
class A {
  public:   _vfptr
    int a;
    void f1();
   virtual void f2();
    void f3();
   virtual void f4();
};
```

```
class B : public A {
  public:
    int b;
    void f4();
    void f5();
  virtual void f6();
};
```

A *ptr=&obj2;



A *ptr=&obj1;

ptr -> f4();

late binding

f4() is 2nd entry in vtable.

① go to object & read first 8 bytes (vfptr) -> it is pointing to vftable.

② go to vftable & access 2nd entry of it

③ call that fn.

Early binding vs Late binding

fun(); → fun() { — — }

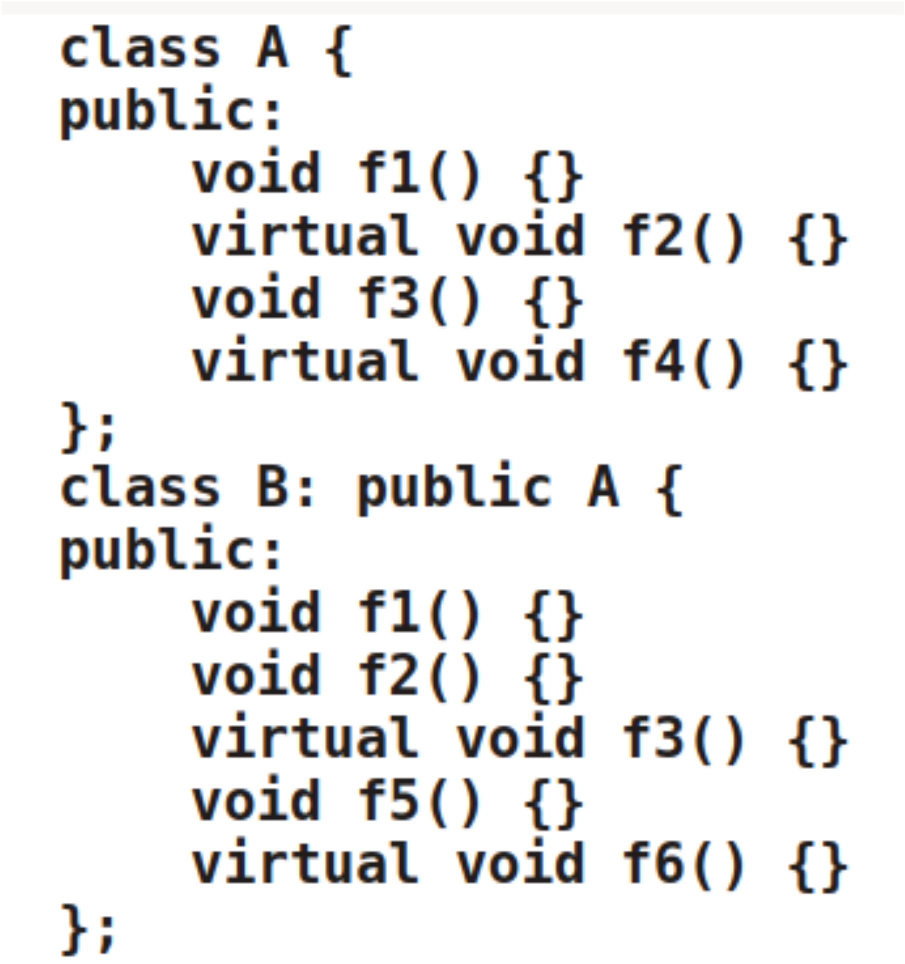
- Virtual function invoked using pointer or reference is always late binding (call is resolved at runtime). Rest all are early binding (call is resolved at compile time).
- In early binding, function is called from the caller's class; while in late binding function is called from object's class.
- In case of late binding execution plan is still prepared at compile time (as vtable is created by the compiler).
  - Find index of function in vftable (n). (2)
  - Go to object address. ✓
  - Read vfptr (first 8 bytes) & get vftable.
  - Call nth function from vftable.

(e) obj.fun();
(e) ref.fun();
(e) ptr → fun();

(e) obj.vfun();
(l) ref.vfun();
(l) ptr → vfun();

A * ptr = NULL;
ptr → f4();

Early binding vs Late binding

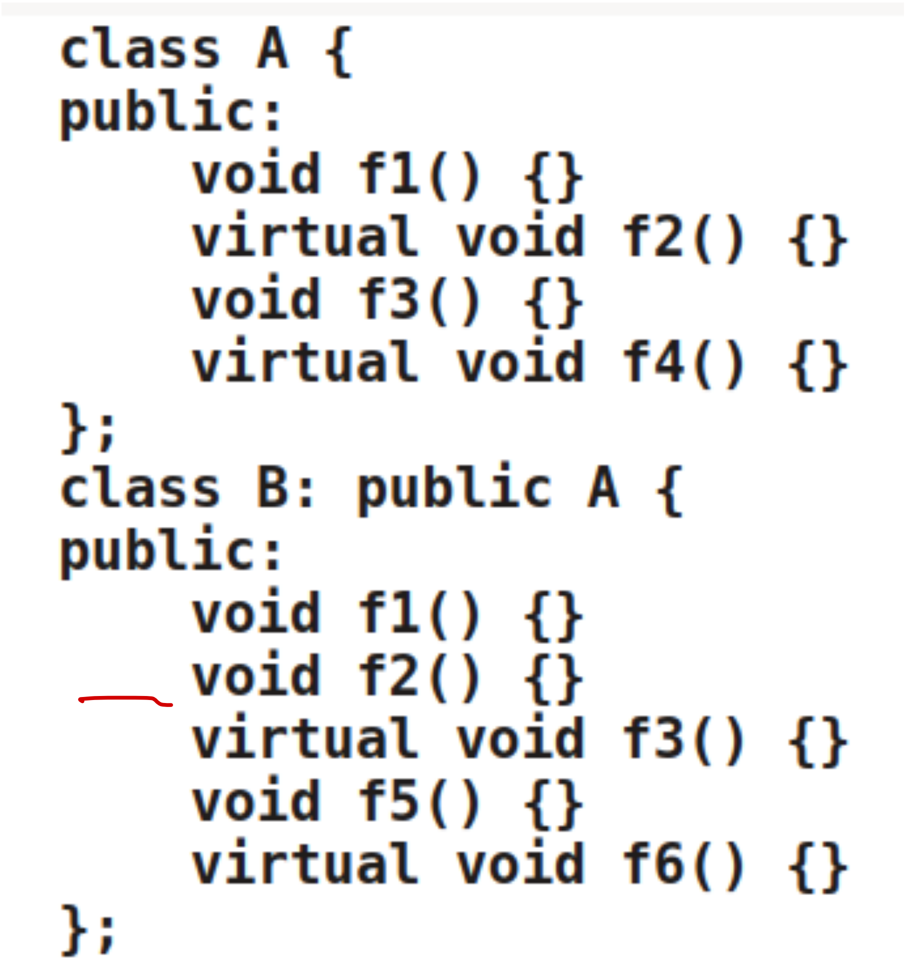


```
class A {
public:
    void f1() {}
    virtual void f2() {}
    void f3() {}
    virtual void f4() {}
};
class B: public A {
public:
    void f1() {}
    void f2() {}
    virtual void f3() {}
    void f5() {}
    virtual void f6() {}
};
```

- **A objA;**
- e **objA.f1();** A::f1
- e **objA.f2();** A::f2
- e **objA.f3();** A::f3
- e **objA.f4();** A::f4
- x **objA.f5();** x
- x **objA.f6();** x

- **B objB;**
- e **objB.f1();** B::f1
- e **objB.f2();** B::f2
- e **objB.f3();** B::f3
- e **objB.f4();** A::f4
- e **objB.f5();** B::f5
- e **objB.f6();** B::f6

Early binding vs Late binding

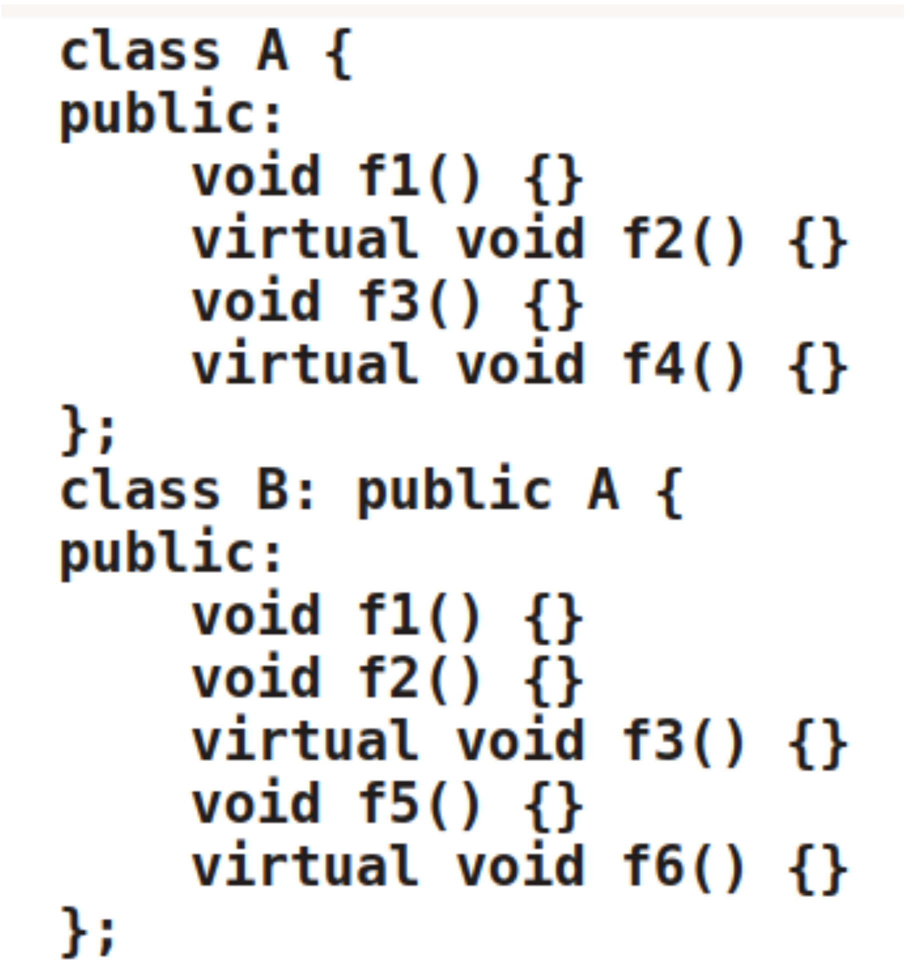


```
class A {
public:
    void f1() {}
    virtual void f2() {}
    void f3() {}
    virtual void f4() {}
};
class B: public A {
public:
    void f1() {}
    void f2() {}
    virtual void f3() {}
    void f5() {}
    virtual void f6() {}
};
```

- A *pA = new A;
- e pA->f1(); A::f1
- l pA->f2(); A::f2
- e pA->f3(); A::f3
- l pA->f4(); A::f4
- x pA->f5(); x
- x pA->f6(); x

- B *pB = new B;
- e pB->f1(); B::f1
- l pB->f2(); B::f2
- l pB->f3(); B::f3
- l pB->f4(); A::f4
- e pB->f5(); B::f5
- l pB->f6(); B::f6

Early binding vs Late binding



```
class A {
public:
    void f1() {}
    virtual void f2() {}
    void f3() {}
    virtual void f4() {}
};
class B: public A {
public:
    void f1() {}
    void f2() {}
    virtual void f3() {}
    void f5() {}
    virtual void f6() {}
};
```

- A *pA = new B;
- c pA->f1(); A::f1
- l pA->f2(); B::f2
- c pA->f3(); A::f3
- l pA->f4(); A::f4
- x pA->f5(); x
- x pA->f6(); x

- B *pB = new A; X error
- pB->f1();
- pB->f2();
- pB->f3();
- pB->f4();
- pB->f5();
- pB->f6();

Virtual constructor & virtual destructor

ptr → vfun(); → run time polymorphism.

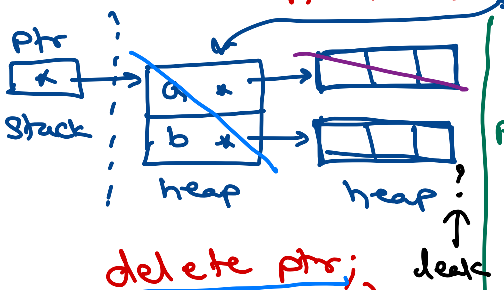
- Virtual constructor is not possible.
  - Virtual functions are designed to called on pointers, but constrctors are called by class name (obj creation).
  - Initialization of vptr itself is done in constructor.
- Virtual destructor is needed in special case.
  - When delete operator is used on base pointer, that points to dynamically allocated derived object, only base class destructor is called. So memory released by derived class destructor is leakaged.
  - This problem is handled by virtual destructor in base class. → So that destructor is called depending on type of object.

ClassName obj; OR ClassName * ptr = new ClassName;

① ctors are never called on pointer, so no point in making them virtual.

② virtual fns are late binded, via vfptr & vftable. but vfptr is initialized into ctor. So it can't be virtual.

A * ptr = new B;

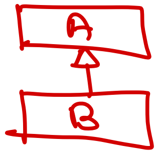


delete ptr;

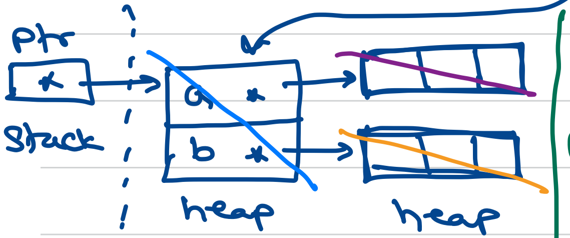
~A()

```
class B : public A {
    int * b;
public:
    B () {
        b = new int[3];
    }
    ~ B() {
        delete [] b;
    }
};
```

```
class A {
    int * a;
public:
    A() {
        a = new int[3];
    }
    ~A() {
        delete [] a;
    }
};
```



A * ptr = new B;



delete ptr;

~B()

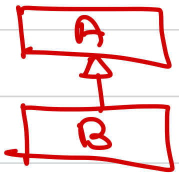
~A()

```
class B : public A {
    int * b;
public:
    B() {
        b = new int[3];
    }

    ~B() {
        delete[] b;
    }
};
```

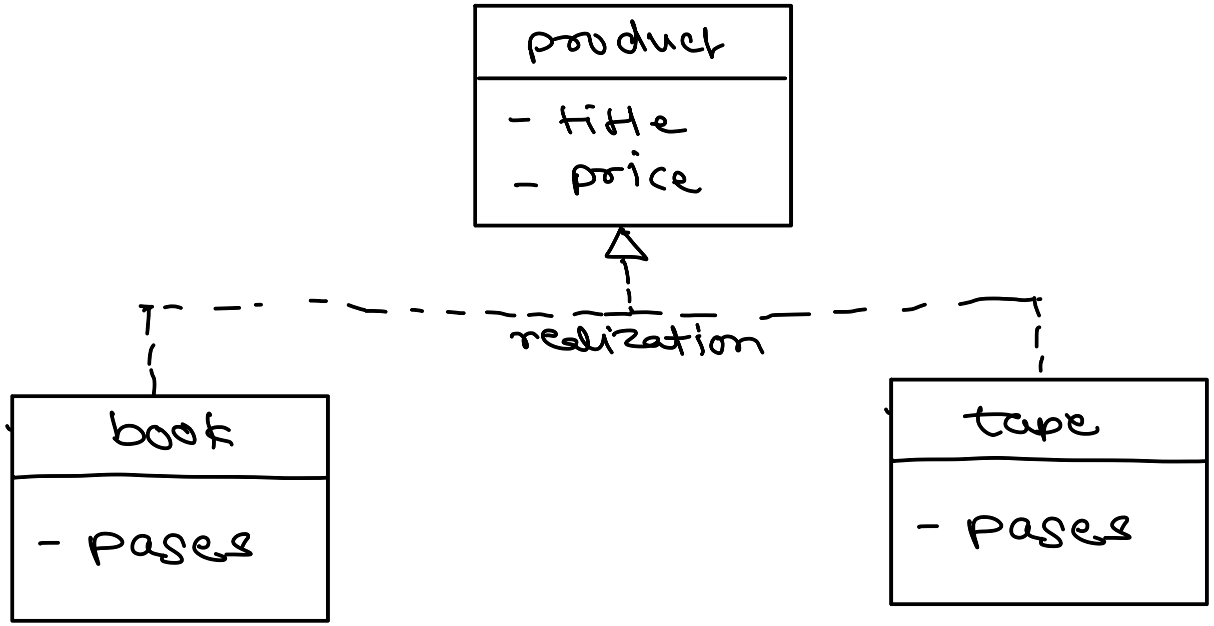
```
class A {
    int * a;
public:
    A() {
        a = new int[3];
    }

virtual ~A() {
        delete[] a;
    }
};
```



Abstract class

- If class needs a method which doesn't have implementation (or partial), then it is declared as "pure virtual" or "abstract" method.
- In C++, pure virtual function declared as
  - virtual void func()=0;
  - 0 indicate NULL entry in vtable.
- Virtual function may not have body.
- Pure virtual fn must be overridden in the derived class; otherwise it remains abstract.
- If class contains atleast one function as pure virtual, then object of such class cannot be created & is referred as "abstract class".

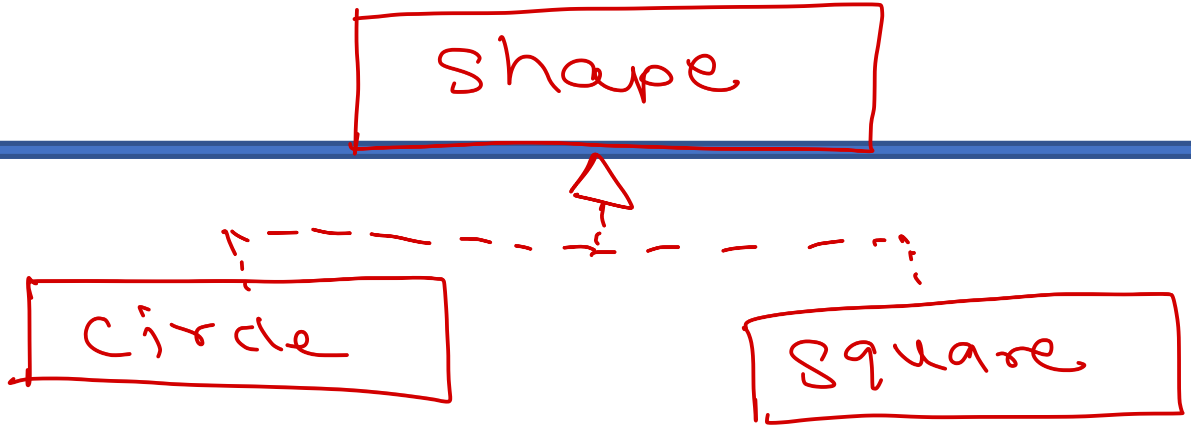


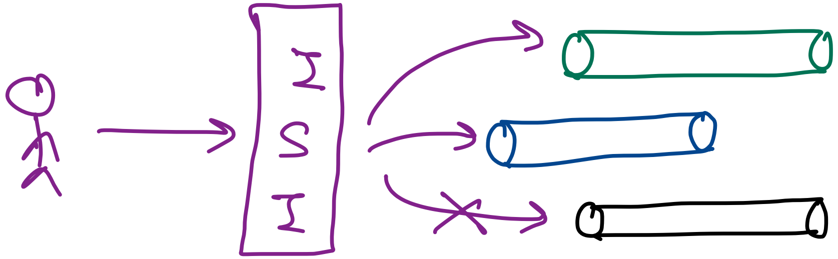
Abstract class

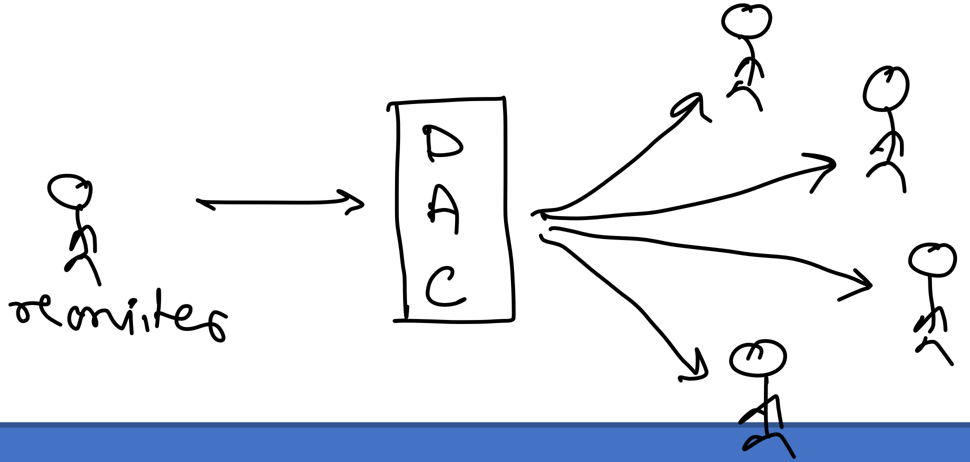
- Abstract class is created, when object of class is not applicable/desirable.
- Abstract class can have data members and other member fns as well (along with pure virtual fns), which can be reused by derived class object.
- They can also have constructor & destructor, which are used to init & de-init its data members. → when derived class object is created/destroyed,
- They force pure virtual functions to be overridden in derived class.
- Usually pure virtual & virtual functions are called on base pointer or reference to achieve runtime polymorphism.

Interface

- If class contains only pure virtual functions (no data members & other functions), then it is called as “interface”.
- They force (all) pure virtual functions to be overridden in derived class.
- These functions are called on base pointer or reference to achieve runtime polymorphism.
- Typically interfaces are used to design specifications/standards.
- Interfaces are immutable & handles fragile base class problem.
- Interfaces can also be used to group different/non-related objects together.







① class

ⓐ data members & member fns → reusable in derived class.

ⓑ virtual fn that can be overriden in derived class and achieve run time polymorphism.

ⓒ can create objs to represent real world entities.

② abstract class

ⓐ data members & member fns → reusable in derived class.

ⓑ virtual fn that can be overriden in derived class and achieve run time polymorphism.

ⓒ pure virtual fn that force derived class to override them.

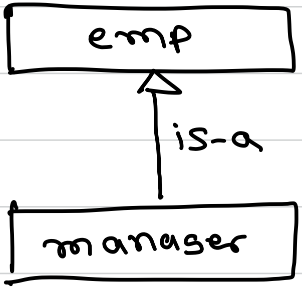
ⓓ cannot create objects (not real world entities).

③ interface

ⓐ all fns are pure virtual, that must be overriden in derived class → design a standard/specification.

ⓑ achieve runtime polymorphism (on interface pointer).

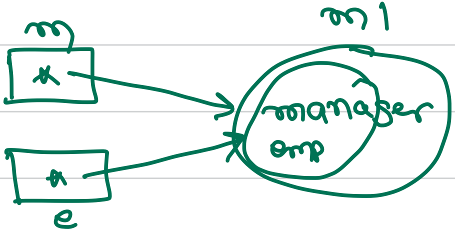
ⓒ cannot create objects (not real world entities).

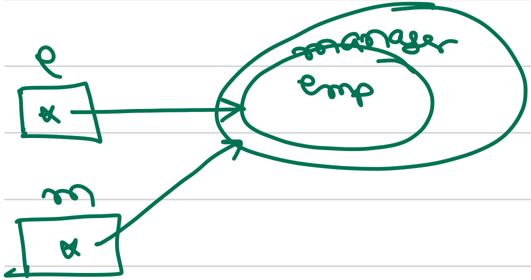


```
manager m1;
manager *m = &m1;
emp * e= m;   ← up-casting
```

↓

object slicing





```
manager  m2;
emp  *e = &m2;
manager *m = (manager*) e;   ← down-casting
```

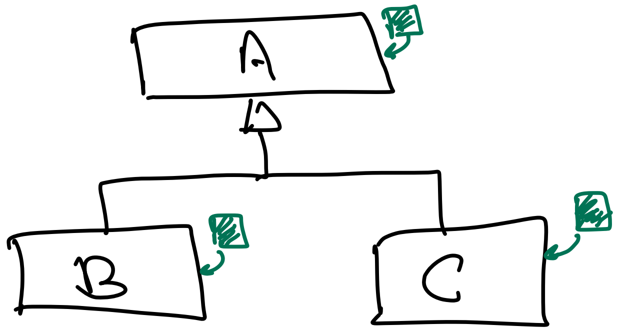
RTTI

- Each C++ class is associated with type_info object that stores information about the class. It contains
  - name()
  - operator==()
  - operator!=()

type_info

- This object is associated with vtable of the class (if class has virtual fn).
- typeid() operator is used to access type_info object.
- Operator function dynamically (run-time) in polymorphic scenario; otherwise it works statically (compile-time).

if class have atleast one virtual function.



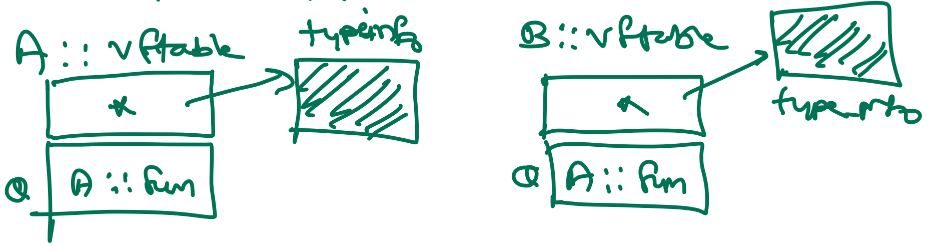
A *p = new B;

const typeinfo &t1 = typeid (p); A*

const typeinfo &t2 = typeid (*p); A

for polymorphic hierarchy

↳ virtual function → virtual table.

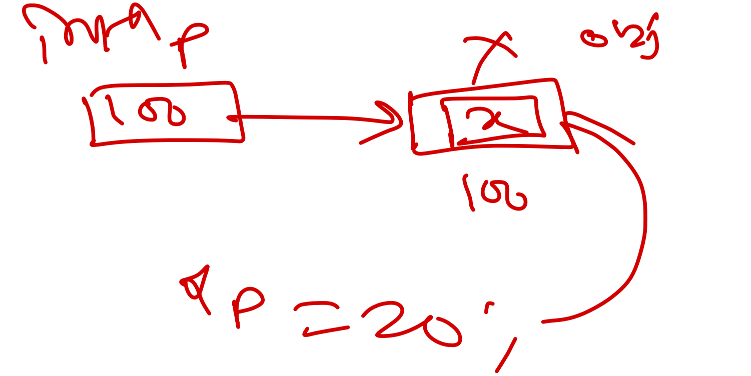


Casting Operators

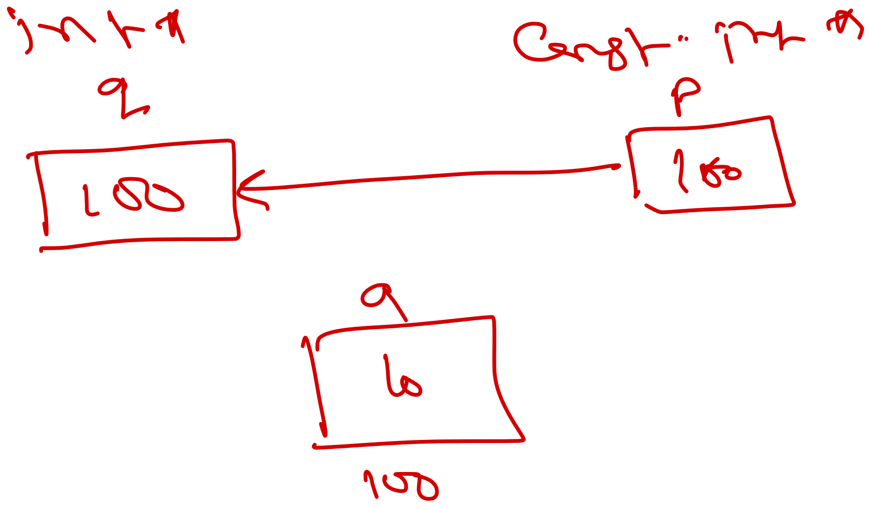
- C style casting works in C++.
- However C++ provides four more casting operators
  - static_cast<>
  - dynamic_cast<>
  - const_cast<>
  - reinterpret_cast<>

mainly used with pointers or references.

- const_cast<> is used to remove const-ness or volatile-ness of pointer or reference temporarily.
- reinterpret_cast<> is used to reinterpret data pointed by pointer in different way. This cast should be used carefully.



can modify private data members.



Casting Operators

- static_cast<> checks inheritance relation between pointer/reference at compile time. It is used for down-casting pointers (statically).
- dynamic_cast<> checks inheritance relation between pointer/reference at run time. It is used for down-casting pointers (dynamically).
- dynamic_cast<> can only be used with polymorphic class i.e. class must contains at least one virtual function.
- If dynamic_cast<> fails at runtime, it returns NULL pointer (while casting pointers) or throws bad_cast exception (while casting references).

```
main() {
    derived obj;
    base *ptr;
    ptr = &obj; ×
ptr = static_cast<derived*>(&obj);
```

check Inheritance relation betn two classes at compile time.

```
}
```

```
class base{
    --
};

class derived :public base{
    --
};
```

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>