Virtual Classes & Polymorphism

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Miri Ben-Nissan
- Version 3: Dr. Erel Segal-Halevi

Example (revisited)

- We want to implement a graphics system
- We plan to have lists of shape. Each shape should be able to draw itself, compute its size, etc.

Solution #3 – hierarchy

```
class Shape { public:
  void draw() const {cout<<'h';}
  double area() const;
  void drawTwice() const {draw(); draw();}
};

class Square: public Shape { public:
  void draw() const {cout<<'q';}
  double area() const;
};

class Circle: public Shape { public:
  void draw() const {cout<<'c';}
  double area() const;
};
```

Solution #3

Now if we write

```
Shape myShapes[2];
myShapes[0] = Circle();
myShapes[1] = Square();
for (…) myShapes[i].draw();
```

What will happen?

Solution #3

Now if we write

```
Shape myShapes[2];
myShapes[0] = Circle();
myShapes[1] = Square();
```

What will happen?

—- The Circle and Square will be constructed and then sliced to fit inside the Shape objects.

“`myShapes[0] = Circle()`” copies from the circle, its hidden “Shape” field.

Solution #3

Now if we write (like in Java):

```
Shape* myShapes[2];
myShapes[0] = new Circle();
myShapes[1] = new Square();
```

What will happen when we call

```
myShapes[0]->draw(); ?
```

Solution #3

Now if we write (like in Java):

```
Shape* myShapes[2];
myShapes[0] = new Circle();
myShapes[1] = new Square();
```

What will happen when we call

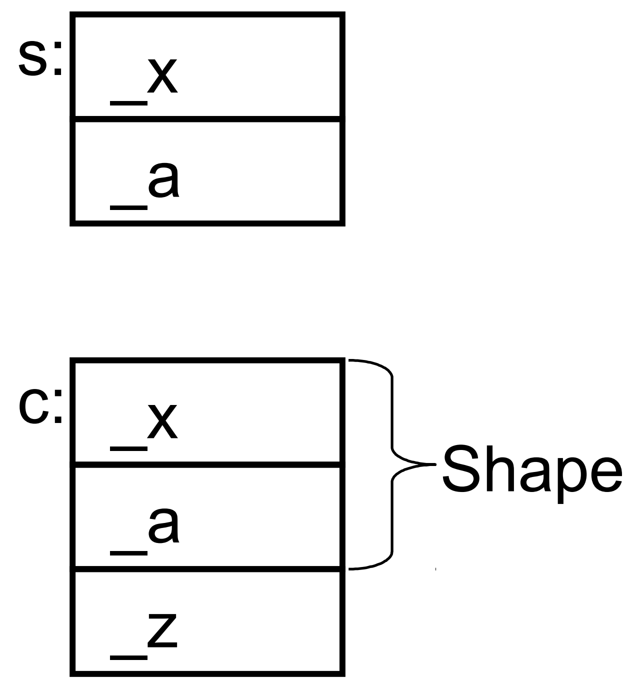
```
myShapes[0]->draw(); ?
```

No slicing, but still, **h will be printed!**

Underneath the Hood: Static Resolution

```cpp
class Shape
{
    double _x;
    int _a;
};

class Circle:
  public Shape
{
    double _z;
};
```

```cpp
Shape s;
Circle c;
```



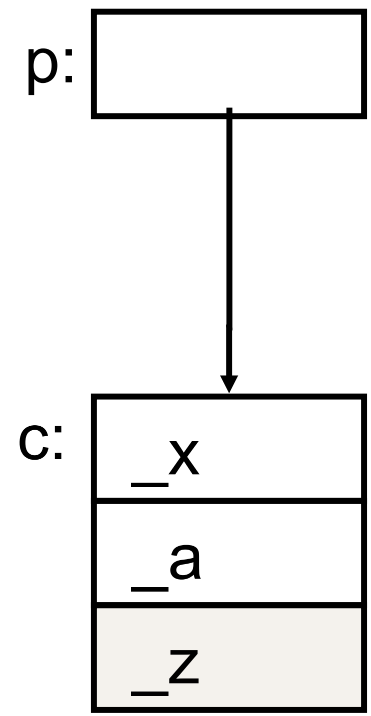
Pointing to an Inherited Class

```
Circle c;
Shape* p = &c;
```

p points to the hidden "Base" field inside d.

When using *p, we treat d as though it was a Base object.

The compiler cannot know if *p is from a derived class or not!



Dynamic Resolution

Static/early resolution

is clearly not what we want to in this example.

- Based on the type of the *variable.*
- Determined at compile time.

Dynamic/late resolution:

is more desirable here:

- Based on the type of the *object*
- Determined at run time

[Java Like]

dynamic resolution

The `virtual` keyword states that the method can be overridden in a dynamic manner.

```
class Shape
{
public:
 virtual void draw() const
    {cout<<'h';}
 virtual double area() const;
};

class Square: public Shape
{
public:
 virtual void draw() const
    {cout<<'q';}
 virtual double area() const;
};
```

```
class Circle: public Shape
{
public:
 void draw() const
    {cout<<'c';}
 double area() const;
};
```

dynamic resolution

Returning to the shapes example, using virtual methods gives the desired result:

```
Shape* s=new Circle;
s->draw();
```

Will print 'c'

Virtual Methods

Class `Base` defines a `virtual` *method* `foo()`

The resolution of `foo()` is dynamic in **all** subclasses of `Base`.

- If the subclass `Derived` overrides `foo()`, then `Derived::foo()` is called
- If not, `Base::foo()` is called

With references

```
struct Base
{
    virtual void f()
    {
        cout << "B" << endl;
    }
};


struct Derived: public Base
{
     virtual void f()
    {
        cout << "D" << endl;
    }
};
```

```
int main()
{
    Derived d;

    Base b = d;
    b.f(); //B

    Base& bref= d;
    bref.f(); //D

    Base b1;
    // Derived d1 = b1;
    // won't compile
}
```

Base function that calls virtual function

```
struct Base {
  virtual void f() { cout<< "Base f()" <<endl; }
          void g() { f(); }
};

struct Derived : public Base {
  void f() { cout<< “Derived f()" <<endl; }
};

int main(){
  Derived d;
  d.g()
```

will print “Derived f()”. Why??

Base function that calls virtual function

```
struct Base {
  virtual void f() { cout<< "Base f()" <<endl; }
          void g(Base* this) {this->f(); }
};

struct Derived : public Base {
  void f() { cout<< “Derived f()" <<endl; }
};

int main(){
  Derived d;
  Base::g(&d)
}
```

will print “Derived f()”. Why??

C++

Calling virtual function from a constructor

```
struct Base {
  Base() { f(); }
  virtual void f(){ cout<<“Base”<<endl;}
};
struct Derived: public Base {
  virtual void f(){ cout<<“Derived”<<endl;}
};
int main(){
  Derived d; // would print “Base”
}
```

Why? Because when Base() is called, Derived is not constructed yet! https://stackoverflow.com/q/962132/827927s

Calling virtual function from a destructor

```
struct Base {
  ~Base() { f(); }
  virtual void f() { cout<<“Base”<<endl;}
};
struct Derived: public Base {
  virtual void f() { cout<<“Derived”<<endl;}
};
int main(){
  Derived d; // would print “Base”
}
```

Why? Because when ~Base() is called, Derived is already destructed! https://stackoverflow.com/q/962132/827927

Polymorphism rules:

When calling a method, polymorphism will take place if:

- We call a method through pointer or reference to a base class that actually points to a derived object.
- The method must be virtual.
- We are not in ctor / dtor
- The derived class must override the base method with exactly the same signature (C++11 put **override** between () and { } to check that the method really overrides in compile time)

Implementation of Virtual Methods (under the hood)

Implementation of Virtual Methods

Possible approach:

- If `foo()` is a virtual method, then each object has a pointer to the implementation of `foo()` that it uses
- Can be implemented by using array of pointers to functions

Cost:

- Each virtual method requires a pointer
  - Large number of virtual methods
  - waste of memory

Implementation of Virtual Methods

Alternative solution:

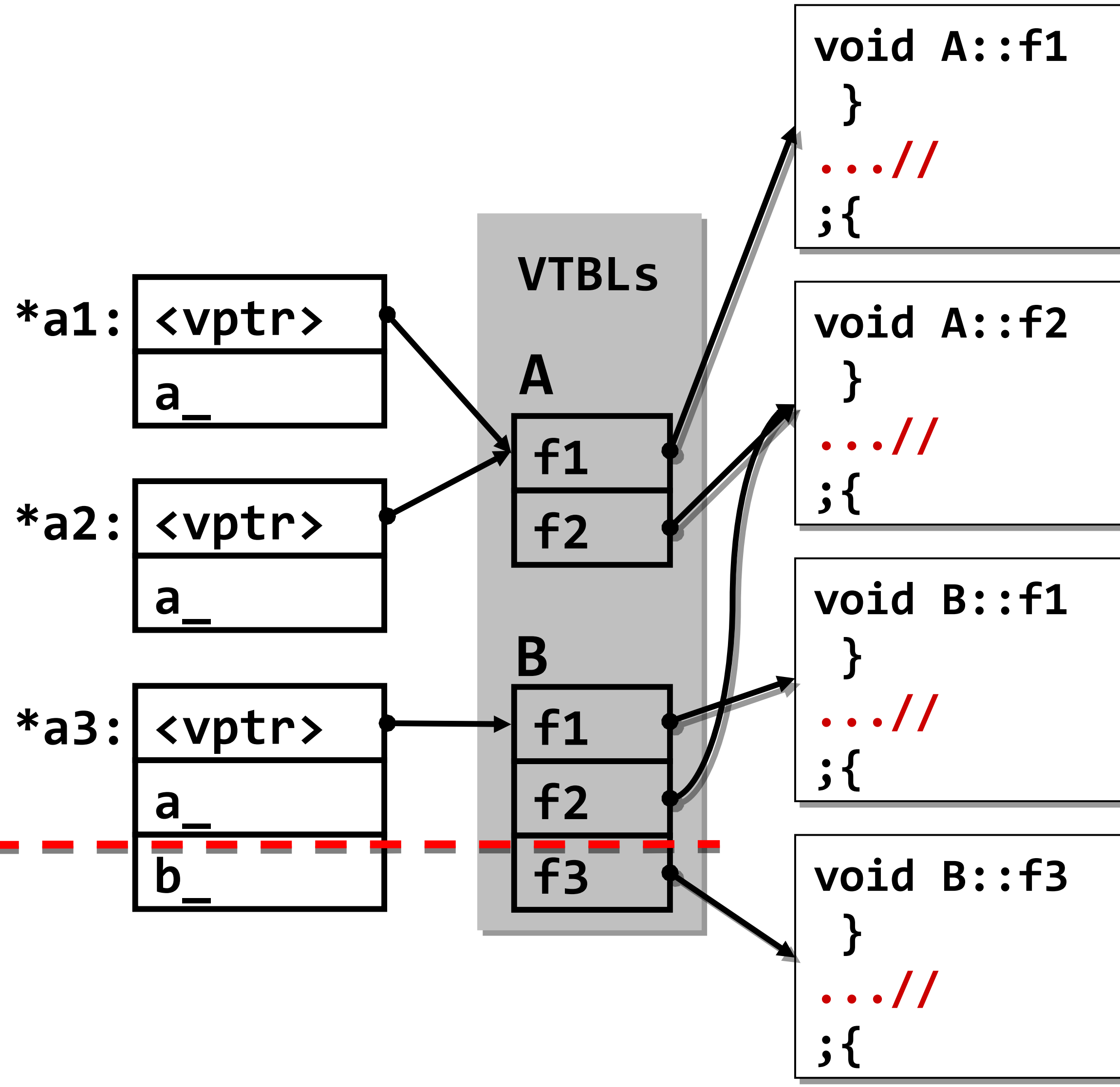
- Each object has a **single** pointer to an array of function pointers
- This array points to the appropriate functions

Cost:

- For each class, we store one table
- Each object contains one field that points to the right table

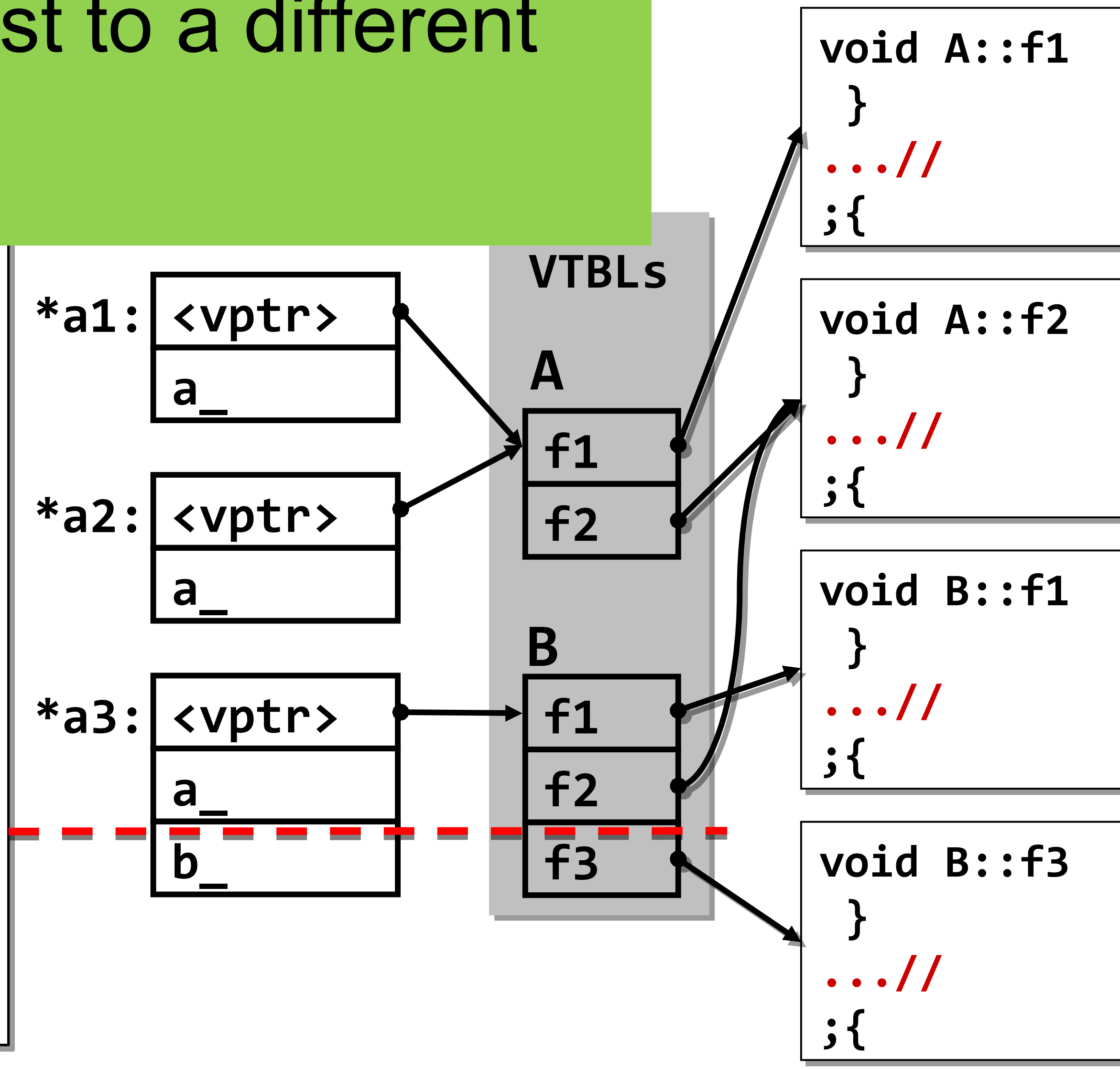
++C

```
class A
{ public:
   virtual void f1();
   virtual void f2();
   int _a;
};

class B: public A
{ public:
   void f1();
   virtual void f3();
   void f4();
   int _b;
};

A* a1= new A;
A* a2= new A;
A* a3= new B;
a1→f1();
a3→f1(); // same code
```



Through *a3 everything below the red dashed line will be hidden (you can downcast to a different name, later)

```
class B: public A
{ public:
   virtual void f1();
   virtual void f3();
   void f4();
   int _b;
};

A* a1= new A;
A* a2= new A;
A* a3= new B;
a3→f3(); // comp.err.
a3→f4(); // comp.err.
```



Virtual Functions - demo

Either view folder 2

Or put the following code in https://godbolt.org/

```
class Base {
    public:
        int x, y;
        int f() { return 111; }
    virtual int g() { return 222; }
    virtual int h() { return 333;}
};

class Derived: public Base {
    int g() { return 444; }
};

int main() {
    Base* p = new Derived;
    p->f();
    p->g();
    p->h();
        return 0;
```

Calling virtual function from a ctor/dtor explained

- When the code to the ctor/dtor is generated, it is generated to its class and not for a different class.
- Thus, the vptr will be to the vtable of the same class.

Virtual – cost

- **Time**: Calling a virtual method is more expensive than standard calls
  - Two pointers are “chased” to get to the address of the function
  - No inlining
- **Memory**: objects with virtual methods have an additional fields (about 8 bytes).
- **Conclusion**: Declare a function “virtual” only if you need polymorphism.

Destructors & Inheritance

```
class Base
{ public:
    ~Base();
};
class Derived : public Base
{ public:
    ~Derived();
};
Base *p = new Derived;
delete p;
```

Which destructor is called?

Destructors & Inheritance

```
class Base
{ public:
    ~Base();
};
class Derived : public Base
{ public:
    ~Derived();
};
Base *p = new Derived;
delete p;
```

Which destructor is called? `Base::~Base()!`

Virtual Destructor

- Destructor is like any other method
- The example uses static resolution, and hence the wrong destructor is called
- To fix that, we need to declare virtual destructor **at the base class!**

Once you declare virtual destructor, derived class must declare a destructor

Destructors & Inheritance

```
class Base
{ public:
    virtual ~Base();
};
class Derived : public Base
{ public:
    ~Derived();
};
Base *p = new Derived;
delete p;
```

Which destructor is called? Derived::~Derived()!

Abstract classes

Revisiting our example, we write:

```
class Shape
{
public:
    virtual ~Shape();
    virtual void draw() const;
    virtual double area() const;
};
```

How do we implement Shape::draw() ?

Inheritance & Interfaces

- In this example, we never want to deal with objects of type Shape
  - Shape serves the role of an interface
- All shapes need to be specific shapes instances of derived classes of Shape.
- How do we enforce this?

Pure Virtual

We can specify that Shape::draw() must be implemented in derived class

```
class Shape {
public:
    virtual ~Shape() {};
    // pure virtuals
    virtual void draw() const = 0;
    virtual double area() const = 0;
    virtual setName() = 0;
};
```

Pure Virtual

We cannot create objects of a Pure Virtual class – that is an object that contains at least one Pure Virtual method

```
Shape* p; // legal
Shape s; // illegal
p = new Shape; // illegal
Circle c; // legal
p = &c; // legal
p = new Circle; // legal
```

Private Pure Virtual

Legal and often used, derived objects must implement but cannot use:

```
class Shape { private:
   virtual void drawImpl()= 0;
    static int g_numDraws;
public:
void draw() const {
   ++g_numDraws;
   drawImpl();
}
static int numDraws() const { return g_numDraws; }
```

Virtual Methods - Tips

1. **If you have virtual methods in a class, always declare its destructor virtual**
2. **Never call virtual methods during construction and destruction**
3. **Use pure virtual classes without any fields to define interfaces**
4. **Use inheritance with polymorphism with care: Be sure that this is the appropriate solution ("is a" relation)**

Interfaces

- To create an equivalent to java interface – declare a base class with all methods pure virtual and no fields.
- Inheritance can be used to hide implementation. But, you will need a factory, and with templates also pimpl pattern (like in C's List).

C++ pimpl

In List.h file:

```
class List {
public:
    virtual void Add()=0;
    virtual ~List(){};
    static List* make();
};
```

In main.cpp:

```
List* L = List::make();
L→Add();
```

In List.cpp file:

```
class ListImpl: public List {
    int* theInts;
    int numInts;
public:
    ListImpl(): theInts
      (new int[…]) {...}
    void Add() { … }
};
List* List::make() {
    return new ListImpl;
}
```