

# 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.

# Class Hierarchy

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

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

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

# Class Hierarchy

Now if we write

```
Shape myShapes[2];
```

```
Circle c;
```

```
myShapes[0] = c;
```

```
myShapes[1] = Square();
```

```
for (...) myShapes[i].draw();
```

What will happen?

# Class Hierarchy

Now if we write

```
Shape myShapes[2];
```

```
Circle c;
```

```
myShapes[0] = c;
```

```
myShapes[1] = Square();
```

What will happen?

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

“myShapes[0] = c” copies from the circle, its hidden “Shape” field.

# Class Hierarchy

Now if we write (like in Java):

```
Circle c;
```

```
Square s;
```

```
Shape* myShapes[2];
```

```
myShapes[0] = &c;
```

```
myShapes[1] = &s;
```

What will happen when we call

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

# Class Hierarchy

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

```
class Shape
```

```
{
```

```
    double x;
```

```
    int y;
```

```
};
```

```
class Circle:
```

```
    public Shape
```

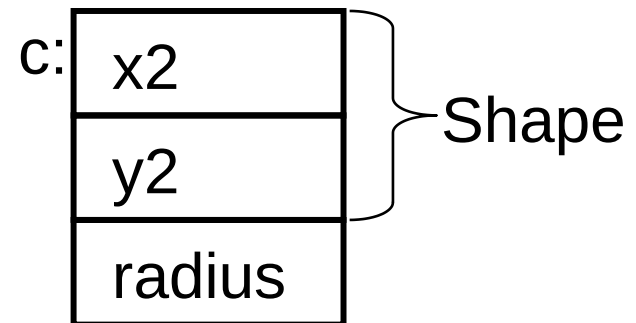
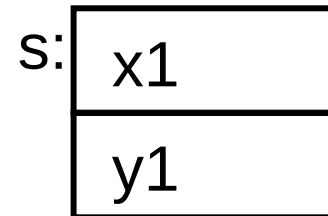
```
{
```

```
    double radius;
```

```
};
```

```
Shape s;
```

```
Circle c;
```





# Pointing to an Inherited Class

```
Circle c;
```

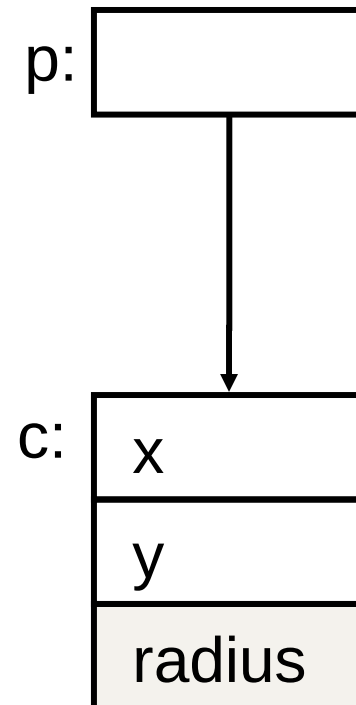
```
Shape* p = &c;
```

p points to the hidden

“Shape” field inside c.

When using \*p, we treat c as though it was a Shape object.

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



# Dynamic Resolution

## Static/early resolution

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

## Dynamic/late resolution:

- 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

# Virtual & references

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

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

```
int main()
{
    Derived d;

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

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

    Base* bp = &d;
    bp->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)  
}
```

...



# 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 in the base.
- 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

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

## Cost:

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

```

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

```

```

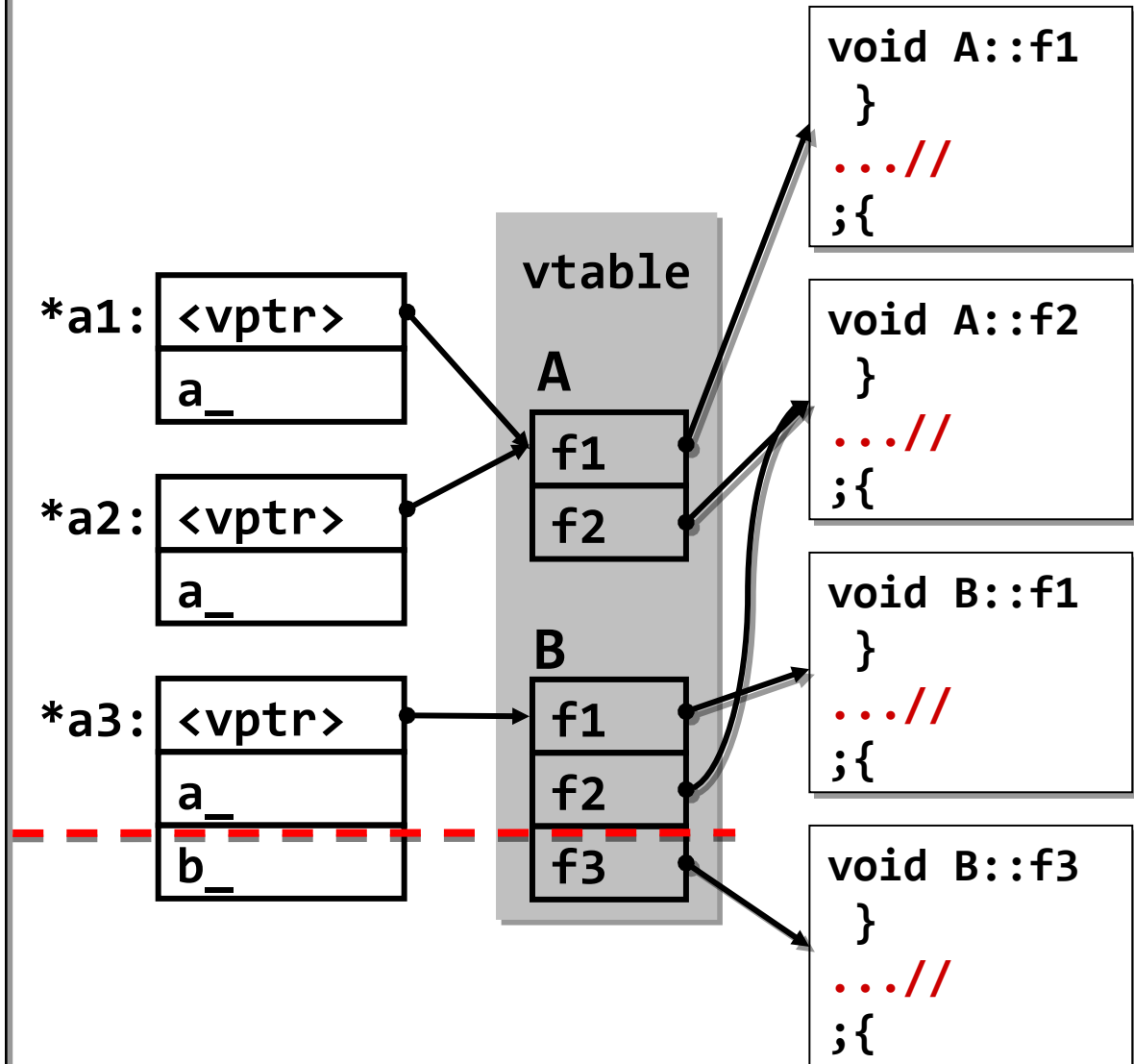
class B: public A
{ public:
    void f1();
    virtual void f3();
    void f4();
    int _b;
};

```

```

A* a1= new A;
A* a2= new A;
A* a3= new B;
//B* b1 = new A;
a1→f1();
a3→f1(); // same code

```

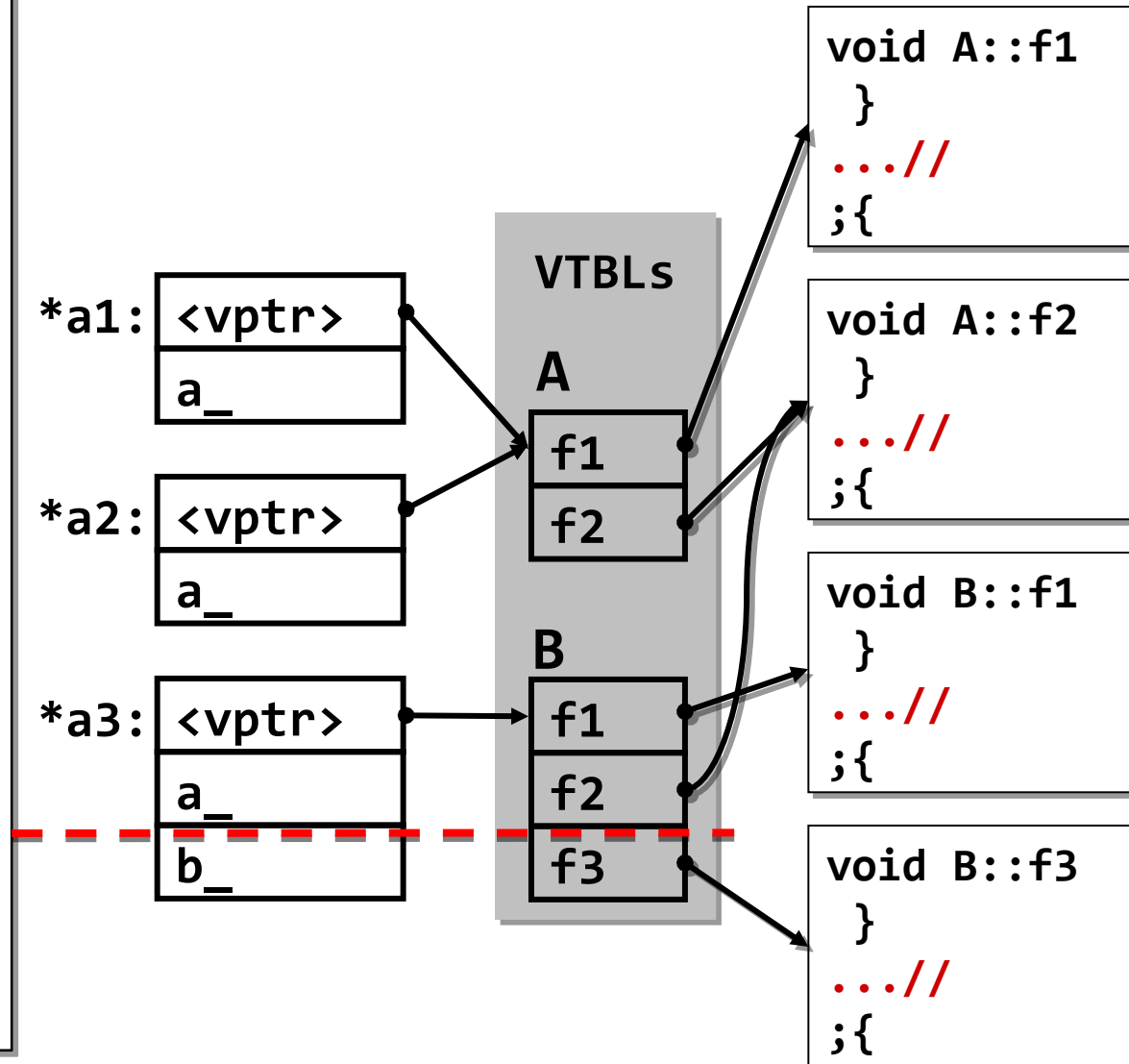


Through \*a3 everything below the red dashed line will be hidden.

```
virtual void f1();
virtual void f2();
int _a;
};
```

```
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.
((B*)a1)->f3();
```



# Virtual Functions - demo

View 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 { public:
    int g() { return 444; }
};

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

# Virtual functions in ctor/dtor - explained

- In the code of `Base::Base`, the `vptr` is set to `Base::vtable`, so the calls are to the `Base` functions.
- Only *after* `Base::Base` is finished, `Derived::Derived` is called and sets the `vptr` to `Derived::vtable`.
- The `vptr` is set in the destructors, too.



## 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.
  - Compiler optimizer cannot inline the call.
- **Memory:** objects with virtual methods have an additional field (~8 bytes).
- **Conclusion:** Declare a function “virtual” only if you need polymorphism.

# Destructors & Inheritance

```
class Base
{ public:
    ~Base() { delete p1; }
};

class Derived : public Base
{ public:
    ~Derived() { delete p2; }
};

Base *p = new Derived;
delete p;
```

*Question: what is the problem here?*

# Destructors & Inheritance

```
class Base
{ public:
    ~Base() { delete p1; }
};

class Derived : public Base
{ public:
    ~Derived() { delete p2; [~Base();] }
};

Base *p = new Derived;
delete p;
```

*Answer: memory leak! Base::~~Base is called.*

# 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!**

# Destructors & Inheritance

```
class Base
{ public:
    virtual ~Base() { delete p1; }
};

class Derived : public Base
{ public:
    ~Derived() { delete p2; [~Base();] }
};

Base *p = new Derived;
delete p;

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

# Pure-virtual (abstract) methods & classes

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

# 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: // pure virtuals:  
    virtual void draw() const = 0;  
    virtual double area() const = 0;  
    virtual void move() = 0;  
    virtual setName(string s) { name = s;}  
    // dtor must have a body  
    // (- it is called by derived dtor):  
    virtual ~Shape() {}  
};
```

# Pure Virtual

```
class Circle: public Shape {  
public:  
    void draw() const { ... }  
    double area() const { ... }  
    void move() { ... }  
};
```

# 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
```

# 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 a pimpl pattern.

# C++ pimpl

## In List.hpp file:

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

## In main.cpp:

```
#include "List.hpp"  
  
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;  
}
```

# 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**