Software Design
CSE 335, Spring 2006

Object-oriented programming: Virtual Functions and Abstract Classes

Stephen Wagner

Michigan State University

Terminology

- Distinction between *operation* and *method*
  - Operations are over-ridden with new methods
  - *print* is an operation; `Employee::print` and `Manager::print` are methods.
- We would like to able to invoke an operation, as opposed to invoking a particular method.

Heterogeneous Containers

- Typically an array is homogeneous. All elements of the array are of the same type.

- If we declare an array of pointers to Shape, the individual elements can be pointers to any class derived from shape.

```
vector<Shape *>shapes(10);
shapes[0]=new Square(0,5,10);
shapes[1]=new Circle(8,10,15);
shapes[2]=new Circle(-5,10,15);
```

Methods versus Operations

- If we want to redraw all the shapes, we want to call the draw *operation* appropriate for each shape.
- For each type of shape, we want to call a different *method*.

Methods versus Operations

- If we want to redraw all the shapes, we want to call the draw *operation* appropriate for each shape.

- For each type of shape, we want to call a different *method*.

```
void Shape::draw()
{
    if (shape_type==CIRCLE)
        draw_circle();
    else if (shape_type==SQUARE)
        draw_square();
```

Virtual Functions

- We want to write functions that invoke an *operation* on an object, not a particular *method*.

- C++ calls these *virtual functions*. Hooking up to a specific instance at runtime is called *dynamic binding*.

Example

```
class Shape
{
    public:
    virtual void draw();
};
```

```
class Circle : public Shape
{
    public:
    void draw();
};
```

Example

```
class Shape
{
    public:
    virtual void draw();
};
```

```
class Circle : public Shape
{
    public:
    void draw();
};
```

```
for (i=0; i<shapes.size(); ++i)
    shapes[i]->draw();
```

Manager Example

```
class Employee {
   private:
     string first_name;
     string last_name;
     Date   hiring_date;
     short  department;
   public:
     virtual void print (ostream & ) const;
};
```

```
class Manager {
   private:
     list<Employee *> group;
     short            level;
   public:
     void print (ostream & ) const;
       };
```

```
void Employee::print(ostream & s) const
{
   s << first_name << " " << last_name " " << endl;
   s << hiring_date << " " << department << endl;
}

void Manager::print(ostream & s) const
{
   Employee::print(s);
   s << "Level: " << level << endl;
}
```

```
int main()
{
    Employee doe("John","Doe",235);
    Manager  howell("George","Howell",200,5);

    doe.print(cout);
    howell.print(cout);

    Employee *ePtr=&howell;
    ePtr->print(cout);
}
```

Terminology

Member Function	Specific function defining some behavior for a class
Method	Member function chosen dynamically based on the type of the object
Operation	A service that is requested from an object instance

Is This Useful?

- In the above examples, we used

```
vector<Shape *>shapes;
```

- What about

```
vector<Shape> shapes;
```

Is This Useful?

- In the above examples, we used

```
vector<Shape *>shapes;
```

- What about

```
vector<Shape> shapes;
```

- We always use pointers when working with virtual functions.

How Does It Work?

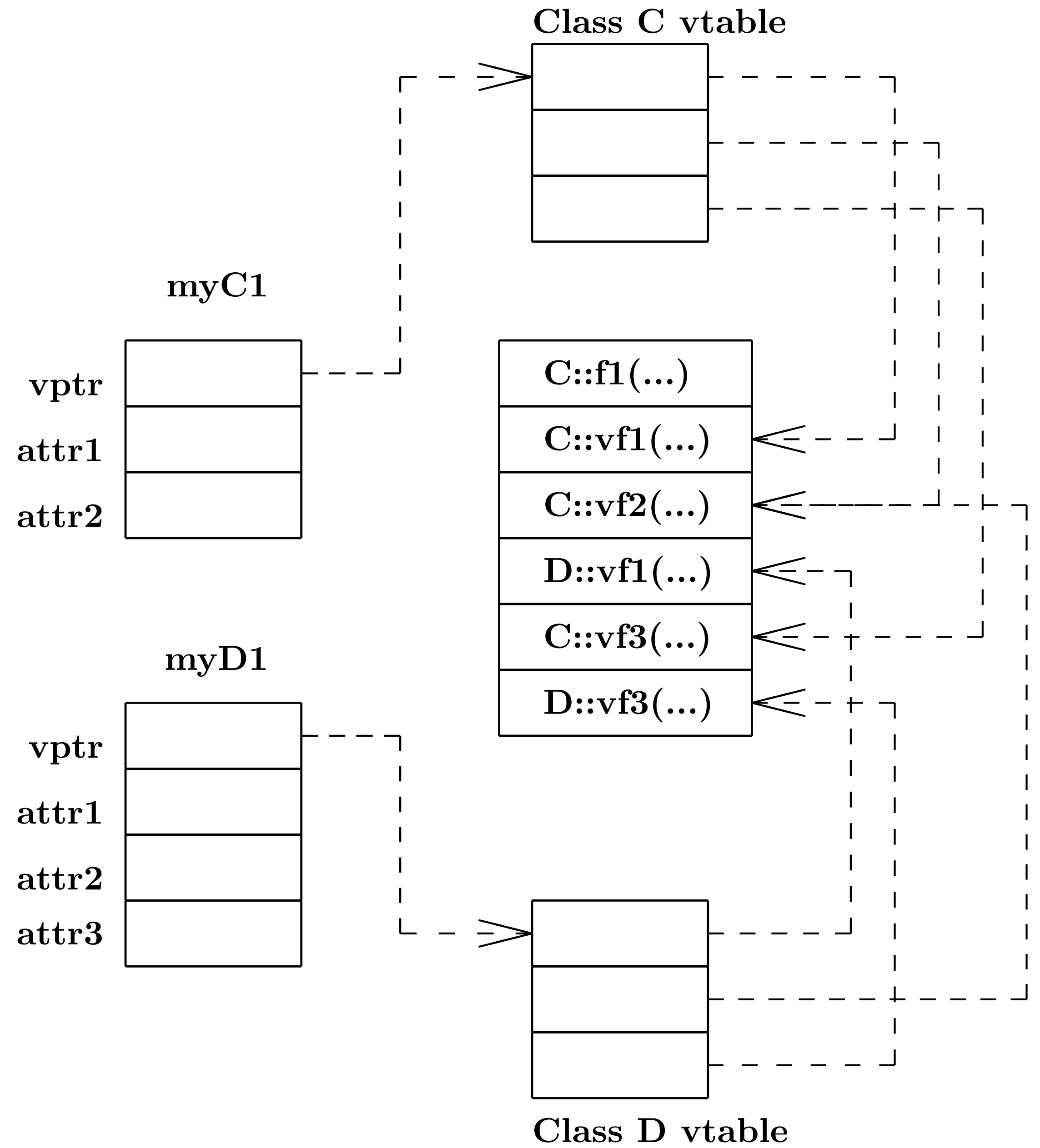
- How are virtual functions actually implemented?

How Does It Work?

- How are virtual functions actually implemented?
- Each object of a class with virtual functions carries with it a pointer to a table of functions (called a vtable)
  - When a virtual function is invoked, the system uses the vtable to determine which function to call
  - Because different objects can have different vtables, the actual function called is determined at runtime

```
class C
{ public:
  void f1(...);
  virtual void vf1(...);
  virtual void vf2(...);
  virtual void vf3(...);
  protected:
  int attr1, attr2; }

class D : public C
{
  void vf1(...);
  void vf3(...);
  protected:
  int attr3; }
```



Code for myPrint (print not virtual)

```
pushl       %ebp
movl        %esp, %ebp

movl        8(%ebp), %eax
pushl       $_cout
pushl       %eax

call        print__8EmployeeR7ostream

movl        %ebp, %esp
popl        %ebp
ret
```

Code for myPrint (print virtual)

```
pushl       %ebp
movl        %esp, %ebp

movl        8(%ebp), %edx
pushl       $_cout
pushl       %edx

movl        4(%edx), %ecx     ;put vptr into ecx
movl        12(%ecx), %eax    ;put address at offest 12
                              ;into eax
call        *%eax             ;call function
                              ;pointed to by eax

movl        %ebp, %esp
popl        %ebp
ret
```

Observation

- Virtual functions do not involve a lot of overhead
- There is one extra level of indirection
  - One array lookup
  - Two extra assembly instructions

Polymorphism

- Virtual functions are an example of *polymorphism*
- *polymorphic:*

Polymorphism

- Virtual functions are an example of *polymorphism*
- *polymorphic:*
  - *poly* =

Polymorphism

- Virtual functions are an example of *polymorphism*
- *polymorphic:*
  - *poly* = many

Polymorphism

- Virtual functions are an example of *polymorphism*
- *polymorphic:*
  - *poly* = many
  - *morphic* =

Polymorphism

- Virtual functions are an example of *polymorphism*
- *polymorphic:*
  - *poly* = many
  - *morphic* = forms (shapes)

Polymorphism

- Virtual functions are an example of *polymorphism*
- *polymorphic:*
  - *poly* = many
  - *morphic* = forms (shapes)
- One name represents many different forms
- Compiler determines correct one at run time

Terminology

- C++ supports three different of polymorphism

Terminology

- C++ supports three different of polymorphism
  - *Ad hoc*: (using function overloading)

Terminology

- C++ supports three different of polymorphism
  - *Ad hoc*: (using function overloading)
  - *Parametric*: (using templates)

Terminology

- C++ supports three different of polymorphism
  - *Ad hoc*: (using function overloading)
  - *Parametric*: (using templates)
  - *Inclusion*: (inheritance and virtual functions)

Terminology

- C++ supports three different of polymorphism
  - *Ad hoc*: (using function overloading)
  - *Parametric*: (using templates)
  - *Inclusion*: (inheritance and virtual functions)
- In OOP, the unqualified term polymorphism refers to inclusion polymorphism
- Classes that define virtual functions are polymorphic types

Examples

Ad Hoc	Show(string &s); Show(char * s); Show(char * s, int length);
Parametric	Find<int>(x); Find<string>(x);
Inclusion	Employee::print(ostream &s); Manager::print(ostream &s);

Destructors

- Consider a vector of pointers to a base class.
- `vector<baseClass *> myObjects;`
- What will happen if we delete the objects?

```
for (unsigned int i=0; i<myObjects.size(); ++i)
    delete myObjects[i]
```

Destructors

- Destructors can be declared virtual
- This will guarantee that the right constructor is called when an object is deleted

```
class A
{
   public:
   A();
   virtual ~A();
};
```

```
class A
{
    public:
    virtual void test();
};
```

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

```
class C : public A
{
    public:
};
```

```
class A
{
    public:
    virtual void test();
};
```

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

```
class C : public A
{
    public:
};
```

```
A * c = new C;
c.test();
```

Virtual Functions

- Derived classes can override virtual functions
- Derived classes do not have to override virtual functions
- The base class defines the “default behavior”

Shape Example

```
class Shape
{
    public:
    virtual double area();
}
```

```
class Square : public Shape
{
    public:
    double area()
     { return length*length; }
};
```

```
class Circle : public Shape
{
    public:
    double area();
     { return PI*radius*radius; }
};
```

Abstract Classes

- There are no shapes
  - Every instance of a shape is really a circle, square, etc.
  - "shape" is an abstraction
- Shape should be an *abstract class*

Abstract Classes

- `Shape::area()` is an abstract operation
- Implemented as a *pure virtual function*

```
class Shape
{
    public:
    virtual double area() = 0;
}
```

Abstract Class

- A class than cannot be instantiated.

Illegal:

```
Shape var;
void f (Shape x);
Shape g();
Shape *s = new Shape;
```

Legal:

```
Shape &var;
void f (Shape & x);
Shape *g();
Shape *s = new Circle;
```

Abstract Classes

- In C++ a class is abstract if
  - It declares or inherits a pure virtual function
  - The constructor is protected

```
class GUIElement
{
   protected:
   int xPosition, yPosition;
   GUIElement(int x=0, int y=0) : xPosition(x),
                 yPosition(y) {};
}
```

Uses of Abstract Classes

- Defing an abstract "placeholder" that can hold objects of various types
  - E.G., `Shape`
  - Useful for building recursive (self-referential) object structures
- Factoring common code into an abstract concept
- Definition of *role-classes* for use in collaboration-based designs

Example

- Design classes for arithmetic expressions
  - variables, literals
  - Unary negation
  - Add, subtract, multiply, etc.

- Each node in the expression should have
  - `evaluate` operation
  - `print` operation

```
class BinaryExpr : public Expr {
   public:
      const Expr* getLeftOperand() const
        {return leftOperand; };
      const Expr* getRightOperand() const
        {return rightOperand; };
   protected:
      const Expr*  leftOperand;
      const Expr*  rightOperand;
      BinaryExpr( const Expr* l, const Expr *r)
         : leftOperand(l), rightOperand(r) {}
};
```