Dynamic binding

Noticing differences between types when it matters

A simple view of the problem

- Suppose we have a class `Circle` derived from a base class `Shape`.
- If we have a pointer or reference to a `Shape`, it might actually be pointing or referring to a `Circle`.
- Why should we care?
- How can we tell?

Why do we care?

- The usual reason is that we want to take one action if the `Shape` is a `Circle` and some other action if it isn't
- Example: rotating a `Circle` requires no action at all

An obvious solution

- Put a type code in each object
- Make sure that the type code is at the same offset in all objects
- Use the type code to decide what to do

The obvious solution can be made to work

- C (and C++) guarantees that if two structures begin with the same sequence of component types, they will have compatible layouts

Implementation (in C)

```
struct Shape {
    int type;
    Point center;
};
struct Circle {
    int type;          // Same as in
    Point center;      // the Shape structure
    int radius;
};
```

Using the type code

```
struct Shape *sp;
/* … */
switch (sp->type) {
case CIRCLE:
    /* … */
    break;
    /* and so on… */
};
```

What's wrong with the simple approach?

- Nothing is wrong with it
  - It can be made to work
- But it does have disadvantages
  - Adding a new type entails changing all the `switch` statements
  - Layout compatibility comes about only through convention
  - The code to deal with `Circles` is scattered all over the place

The C++ approach: virtual functions

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

The function definitions look normal

```
void Shape::draw() {
    // …
}
void Circle::draw() {
    // …
}
```

Calling a virtual function

- When a pointer (or reference) to a base class actually points (or refers) to a derived class object, *and*
- You use that pointer (or reference) to call a function that is declared virtual in the base class, then
- The derived-class function is the one that is actually called.

Examples

```
Shape s;  Circle c;
Shape* sp;  Circle* cp;
Shape& sr = /* something */;
Circle& cr = /* something */;
s.draw();        // Shape::draw
c.draw();        // Circle::draw
sp->draw();      // depends on the object
cp->draw();      // depends on the object
sr.draw();       // depends on the object
cr.draw();       // depends on the object
```

A virtual call happens when

- A function is virtual in the base class
- A pointer or reference to a base class actually points or refers to a derived class object

Typical implementation

- Every object of a type with one or more virtual functions includes a pointer to a *virtual function table*
- Every virtual call fetches the address of the function from a known offset (fixed at compile time) in the table
- Typical cost: a few memory references per call

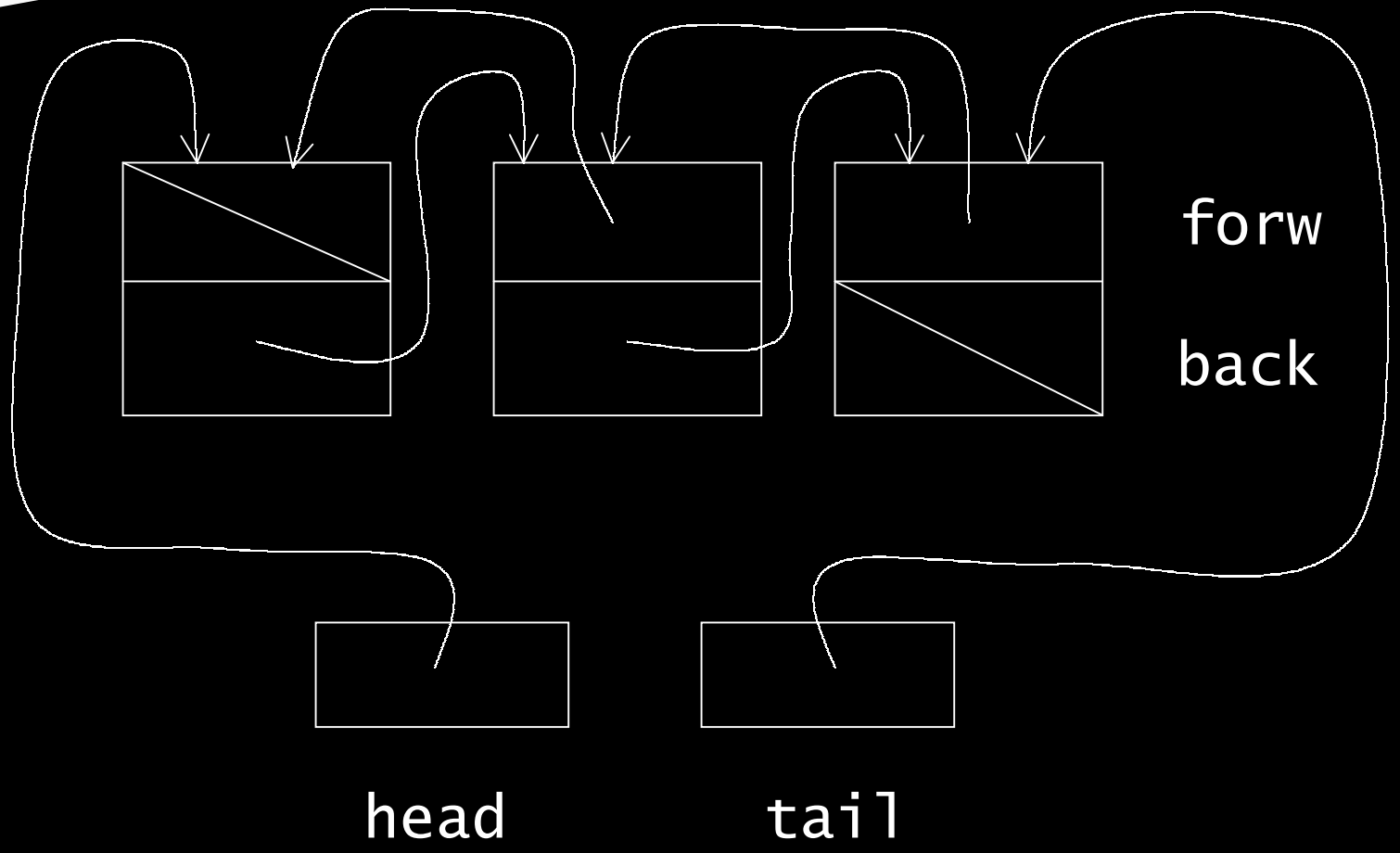
Types of virtual functions

- The argument types must be identical in base and derived classes
- The result types too, unless
  - The base class function returns a pointer (or reference) to some type *T*, and
  - The derived class function returns a pointer (or reference) to a type derived from *T*

An example

- We might have every Shape in the universe put itself on a doubly-linked list
- Then we could easily draw all the Shapes, even if some of them were really objects of classes derived from Shape

The data structure



Example code, part 1

```
Class Shape {
public:
    Shape();
    virtual ~Shape();
    virtual void draw();

private:
    Shape* forw;
    Shape* back;
    // …
};
```

Code, part 2

```
Shape* head = 0;
Shape* tail = 0;
Shape::Shape()
{
    forw = tail;
    back = 0;
    (tail? tail->back: head) = this;
    tail = this;
}
```

Code, part 3

```
Shape::~Shape()
{
  (this==head?head:forw->back) = back;
  (this==tail?tail:back->forw) = forw;
}
```

Adding new shapes

- Just do it…

```
class Circle: public Shape {
public:
    virtual void draw();
// …
};
void Circle::draw()
{ /* … */ }
```

Draw all the shapes

```
void drawall()
{
    Shape* p = head;
    while (p) {
        p->draw();  // virtual call
        p = p->back;
    }
}
```

Why the virtual destructor?

- Whenever
  - You say `delete p`, *and*
  - The type of `p` is "pointer to base," *and*
  - `p` actually points at a derived object
- Then the base class must have a virtual destructor, even if it does nothing

What does a virtual destructor do?

- It is a signal to the compiler that using `delete` (which always destroys the object) should go through the virtual call mechanism
- It has no effect otherwise

Multiple abstractions

- A `Shape` is something that can go on the list defined by `head` and `tail`
- A `Shape` is something that supports the `draw` operation
- A `Circle` is a kind of `Shape` whose `draw` operation is implemented in a particular way

Virtual functions and type fields

- You can use virtual functions to implement type fields:

```
enum Kind { SHAPE, CIRCLE /* … */ };
class Shape {
public:
    virtual Kind my_type() {
        return SHAPE;
    }
    // …
};
```

- But it's often unnecessary in practice

Virtual functions and constructors

- While an object is under construction or destruction, its type is what it was declared to be:

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

Another example

- Suppose we want to represent expressions as trees
- An expression is
  - an integer, or
  - a unary operator applied to an expression, or
  - a binary operator applied to two expressions
- We would like to be able to create and print expressions

Sample code

```
IntExpr* three = new IntExpr(3);
IntExpr* four = new IntExpr(4);
IntExpr* five = new IntExpr(5);
UnaryExpr* negfive =
    new UnaryExpr("-", five);
BinaryExpr* twelve =
    new BinaryExpr("*", three, four);
BinaryExpr* seven =
    new BinaryExpr("+", negfive, twelve);

seven->print(cout);
```

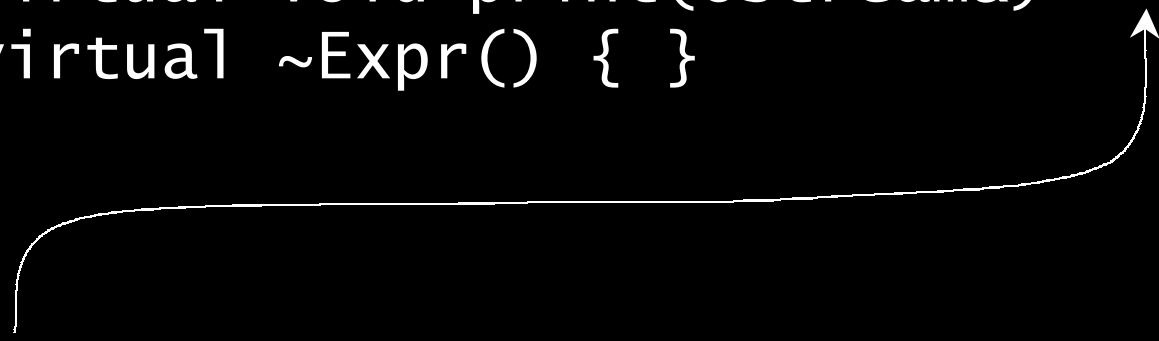
should print ((-5)+(3*4))

How do we do it?

- We will define a base class called `Expr` to represent expressions
  - An `IntExpr` will be a kind of `Expr`
  - as will a `UnaryExpr` and `BinaryExpr`
  - Every kind of `Expr` will support a virtual `print` operation

We can already write code

```
class Expr {
public:
    virtual void print(ostream&) = 0;
    virtual ~Expr() { }
};
```



This makes it a *pure virtual function*

Integer expressions

```
class IntExpr: public Expr {
public:
  IntExpr(int n0): n(n0) { }
  void print(ostream& s) {
    s << n;
  }
private:
  int n;
};
```

Unary expressions

```
class UnaryExpr: public Expr {
public:
  UnaryExpr(const char* s, Expr* e0):
    op(s), e(e0) { }
  void print(ostream& s) {
    s << “(” << op;
    e->print(s);
    s << “)”;
  }
  ~UnaryExpr() { delete e; }
private:
  Expr* e;
  const char* op;
};
```

Binary expressions

```
class BinaryExpr: public Expr {
public:
  BinaryExpr(const char* s, Expr* e01,
    Expr e02): op(s), e1(e01), e2(e02) { }
  void print(ostream& s) {
    s << “(”; e1->print(s); s << op;
              e2->print(s); s << “)”;
  }
  ~BinaryExpr() { delete e1; delete e2; }
private:
  const char* op;
  Expr* e1;
  Expr* e2;
}
```

We can generalize our sample

```
Expr* three = new IntExpr(3);
Expr* four = new IntExpr(4);
Expr* five = new IntExpr(5);
Expr* negfive = new UnaryExpr("-", five);
Expr* twelve =
    new BinaryExpr("*", three, four);
Expr* seven =
    new BinaryExpr("+", negfive, twelve);

seven->print(cout);
```

We can get rid of most of the variables:

```
Expr* e =
    new BinaryExpr("+",
        new UnaryExpr("-", new IntExpr(5)),
        new BinaryExpr("*", new IntExpr(3),
                            new IntExpr(4)));

e->print(cout);
delete e;
```

Points to remember

- Virtual functions are meaningful only in the context of pointers or references
- Pure virtual functions are useful when you know that base class objects will not exist by themselves
- If your class has a virtual function, it probably needs a virtual destructor

Why aren't all C++ member functions virtual?

- Not every class needs inheritance
- The overhead, although small, exists
- Sometimes functions shouldn't be virtual (for example, `operator[]` in the `Vector` example from last lecture)

Summary

- Inheritance makes it easier to describe a family of types by describing their similarities and differences
  - The similar parts go in base classes
  - Each set of relevant differences gets its own derived class
- Virtual functions are an efficient way of recovering the differences in C++

Homework (due Monday)

- Rewrite the `Expr` class hierarchy so that it doesn't use virtual functions or type fields
- The idea is to simulate the virtual-function tables