Virtual Methods

Handout comes to us by way of Andy Maag and Julie Zelenski.

When we left off last time, we were able to derive new types from existing types, and we could upcast from a derived type to its base type. This was useful because it allowed us to share some code. However, it really didn't give us the polymorphism we want to extract from similar types. For example, suppose we had the `Shape`, `Circle`, and `Rectangle` classes from the inheritance handout:

```
class Shape {
   ...
   void draw(void) const;
   ...
};

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

class Rectangle : public Shape {
   ...
   void draw(void) const;
   ...
};
```

Suppose we declared an array of shapes like this:

```
Shape* shapeList[3];
shapeList[0] = new Circle(...); // Upcast to Shape*
shapeList[1] = new Square(...); // Upcast to Shape*
shapeList[2] = new Rectangle(...); // Upcast to Shape*
```

We might want a function to draw all of our shapes in the shape list:

```
void drawShapes(int numShapes, Shape* shapeList[])
{
   for (int i = 0; i < numShapes; i++)
      shapeList[i]->draw(); // Always calls Shape::draw()
}
```

Even though we created a `Circle` object, a `Rectangle` object, and a `Square` object, `drawShapes` only sees them as generic `Shape` objects. There are different ways of solving this problem, and we'll look at a couple of them.

Type Fields

One way to solve this polymorphism problem is to manually store the type within every `Shape`, `Circle`, or `Rectangle` object. We could do this by defining an enumerated type, and marking each object in its constructor with its proper type. However, this is very error prone and tends to lead to a lot of switch statements and casting throughout your code. The `drawShapes` function would then look something like this:

```
void drawShapes(Shape* shapeList[], int numShapes)
{
   Rectangle* rect;
   Circle* circle;
   Square* square;

   for (int i = 0; i < numShapes; i++) {
      switch(shapeList[i]->type) {
         case eRectangle:
            Rectangle* rect = static_cast<Rectangle *>shapeList[i];
            rect->draw();
            break;
         case eSquare:
            square = static_cast<Square*>shapeList[i]; // Downcast
            square->draw();
            break;
         case eCircle:
            circle = static_cast<Circle*>shapeList[i]; // Downcast
            circle->draw();
            break;
      }
   }
}
```

Static vs. Dynamic Type

So far, we haven't been able to break the bounds of the declaration type, or the **static type**, of an object. The declared type of an object or an object pointer has been determining which method will be called. The compiler pays no attention to the true type of the object — which is the type we provided when we allocated it with `new`. This is also called the object's **dynamic type**.

Virtual Functions

The way to retain the behavior of the object's instantiated type is through the use of `virtual` functions. To declare a method to be a `virtual` function, you simply use the `virtual` keyword when declaring the method in the class definition. When you call a function, it will check the dynamic type of the object before choosing which function to call—this process is called **reification**. For example, we could declare our `Shape`, `Circle`, and `Rectangle` classes like this:

```
class Shape {
   ...
   virtual void draw(void) const;
   ...
};

class Circle : public Shape {
   ...
   virtual void draw(void) const;
   ...
};

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

Now we get the polymorphic behavior we want:

```
Shape* shapeList[maxShapes];
shapeList[0] = new Circle(...); // Upcast to Shape*
shapeList[1] = new Square(...); // Upcast to Shape*
shapeList[2] = new Rectangle(...); // Upcast to Shape*
shapeList[0]->draw(); // Calls Circle::draw()
shapeList[1]->draw(); // Calls Square::draw()
shapeList[2]->draw(); // Calls Rectangle::draw()
```

It is important that you declare the function to be virtual throughout your class hierarchy, or its behavior will be quite unexpected. A virtual method can call a non-virtual method and vice-versa. Overloaded operator functions can be virtual functions, but `static` methods can't be.

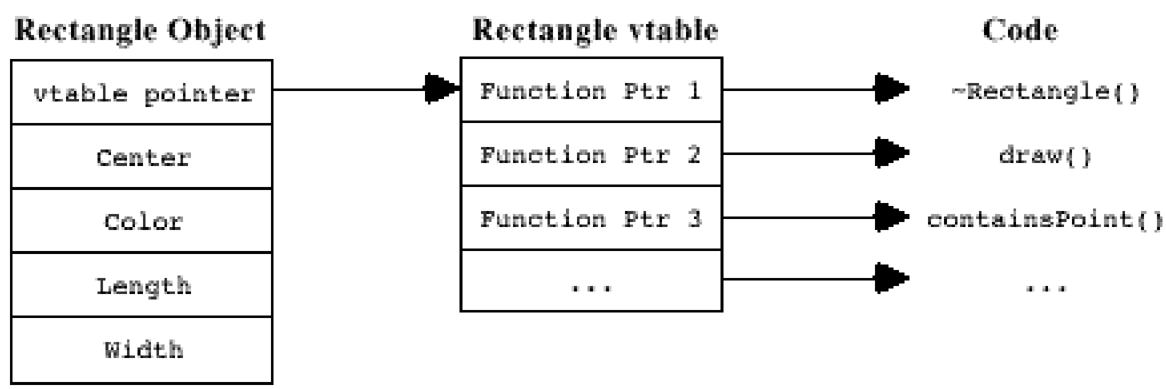
A constructor cannot be a `virtual` function because it needs to know the **exact** type to create. However, a destructor can be declared as `virtual`, and generally should be. `virtual` functions may not seem significant at first, but they enable a ton of code reuse when using class hierarchies. und `Circle`, `Shape`, and `Rectangle` objects, the client doesn't have to as long as all of the relevant functions are `virtual`.

Behind the Scenes

It is sometimes easier to picture how virtual functions work by seeing how they're implemented. You may be surprised to find out that the compiler can't be certain of the dynamic type of an object. From the simple examples above, that seems surprising, but consider this case:

```
Shape *shape = (RandomChance(0.5)) ? new Circle(...) : new Rectangle(...);
shape->draw(); // Should we call Circle::draw() or Rectangle::draw()?
```

The dynamic type of an object cannot be determined statically—that is, the compiler can't always determine what the dynamic type of an object will be at compile-time. Because of this, the compiler has to provide some type of run-time method invocation support. Most C++ implementations use what is called a **virtual function table**, or **vtable** for short. Each class has a vtable which contains a pointer to each of its `virtual` methods. The compiler allocates a hidden pointer, usually at the beginning of an object, which points to its vtable. This pointer is assigned when the



object is allocated, just before the constructor is called. In memory, a `Rectangle` object with a vtable pointer looks something like this:

Each slot in the vtable has an index. When the compiler goes to call the `draw` method, it generates code to 'call `virtual` function 2', as opposed to simply generating code to make a direct call to `Rectangle::draw`.

Efficiency Concerns

Calling a `virtual` method is generally slower than calling a normal method because the vtable pointer must be dereferenced in order to find the address of the function to call. In addition, each object which contains at least one `virtual` function must have an additional pointer allocated within it, so this may impose some memory efficiency concerns if you plan on allocating lots and lots of objects with a vtable.

In general, if you're going to derive from a class, all the functions in that class should `virtual`. If you're absolutely certain you'll never ever ever want to override a particular function, you can make it a non-`virtual` function. However, be warned — forgetting to define a method as virtual can cause hours upon hours of debugging headaches.

Sample Trace

Consider the following source code. Assume that it is compiled using a standard C++ compiler (no default `virtual` behavior)

```
class A {
   public:
      XXXXXXXX void Sketchy()
      {
         cout << "A's Sketchy()" << endl;
         Sketchy(-1);
      }

      YYYYYYYY void Sketchy(int num)
      {
         cout << "A's Sketchy(int) " << num << endl;
      }
};

class B : public A {
   public:
      void Sketchy()
      {
         cout << "B's Sketchy()" << endl;
         Sketchy(-2);
      }

      void Sketchy(int num)
      {
         cout << "B's Sketchy(int) " << num << endl;
      }
};

class C : public B {
   public:
      void Sketchy(int num)
      {
         cout << "C's Sketchy(int) " << num << endl;
      }
};

void Curious(A* wacky)
{
   wacky->Sketchy();
   ((C *)wacky)->Sketchy(123);
}
```

```
void main()
{
   A* inky = new B;
   inky->Sketchy();
   inky->Sketchy(23);
   B* pinky = new C;
   pinky->Sketchy();
   pinky->Sketchy(96);
   Curious(pinky);
}
```

- Suppose the `XXXXXXXX` is replaced with the keyword "`virtual`" and `YYYYYYYY` is replaced with white space. What is the output of this program?
- Suppose the `YYYYYYYY` is replaced with the keyword "`virtual`" and `XXXXXXXX` is replaced with white space. What is the output of this program

Another example: Plain ol Employees and Bosses (courtesy of Julie Zelenski)

Employee.h

```
class Employee {
public:
   Employee(char *nm, float att, float perHour);
   virtual ~Employee();

   // getter/setters for hours, attitude, name, wage?
   // not listed here to reduce clutter

   virtual float productivity() const;
   virtual float groupProductivity() const;
   virtual float askSalary() const;
   virtual void print() const;

   Employee& operator=(const Employee& src);
   Employee(const Employee& src);

private:
   void copyContents(const Employee& src);

   char *name;
   float attitude, wage;
   int hours;
   static const int fullTime = 40;
};
```

Employee.cc

```
Employee::Employee(char *nm, float att, float perHour)   // Constructor
   : name(CopyString(nm)), wage(perHour), attitude(att), hours(fullTime)
{}    // empty constructor body


Employee::~Employee() // Destructor
{
    delete[] name;
}

/*
 * copyContents helper (private)
 * ----------------------------
 * Used by both copy constructor and op= to copy values from
 * one Employee to another.  Does a true deep copy, including
 * making a new copy of the name string
 */

void Employee::copyContents(const Employee& src)
{
    name = CopyString(src.name);
    attitude = src.attitude;
    hours = src.hours;
    wage = src.wage;
}
```

```
Employee& Employee::operator=(const Employee& src)
{
    if (this != &src) { // check for self-assignment
      delete[] name;
      copyContents(src);
    }
    return *this;
}


Employee::Employee(const Employee& src)
{
    copyContents(src);
}


float Employee::productivity() const
{
    return hours * attitude;
}


float Employee::groupProductivity () const
{
    return productivity();
}


float Employee::askSalary() const
{
    return hours*wage;
}


void Employee::print() const
{
    cout << "Name: " << name << endl;
    cout << "Salary: $" << askSalary() << endl;
}
```

Boss.h

```
class Boss: public Employee {

public:
   Boss(char *nm, float att, float perHour, char *ttl);
   virtual ~Boss();

   void setUnderling(Employee *u) { underling = u;}
   Employee *getUnderling() const {return underling;}

   virtual float groupProductivity() const;
   virtual float askSalary() const;
   virtual void print() const;

   Boss& operator=(const Boss& src);
   Boss(const Boss& src);

private:
   Employee *underling;
   char *title;
};
```

Boss.cc

```
Boss::Boss(char *nm, float att, float perHour, char *ttl)
   : Employee(nm, att, perHour), title(CopyString(ttl)), underling(NULL)
{} // explicitly call parent constructor in ctor init list


Boss::~Boss()
{
     delete[] title; // no explicit call to parent destructor
}

Boss::Boss(const Boss& src)                    // copy constructor
 :  Employee(src), title(CopyString(src.title)), underling(src.underling)
{}

Boss& Boss::operator=(const Boss& src)       // assignment operator
{
    if (this != &src) {
      Employee::operator=(src);  // use Employee= to copy employee part
      delete[] title;
      title = CopyString(src.title);   // now copy our extra Boss fields
      underling = src.underling;
    }
    return *this;
}

float Boss::askSalary() const
{
    return Employee::askSalary() * 2;
}
```

```
float Boss::groupProductivity() const
{
    float uProd = (underling != NULL ? underling-> groupProductivity() : 0.0);
    return productivity() + uProd;
}

void Boss::print() const
{
    Employee::print();  // do usual Employee print behavior
    cout << "Title: " << title << endl;      // add our title at end
}
```

Some points about inheritance

Here I will try to re-state the points from lecture about the details of inheritance.

Compile-time type versus run-time type. Consider the parameter to this function:

```
void Apple(Employee *e)
{
   e->print();
}
```

The compile-time type of the parameter is exactly `Employee*`. But when a variable is declared as a `Employee*` (or `Employee&`), that doesn't guarantee that at run-time it points to exactly an `Employee`, but it will be something that is *at least* an `Employee`. At run-time, the pointer could point to an `Employee` or a `Boss` or any `Employee` subclass, and the run-time type can be different for different invocations of the function.

Due to compile-time type checking, within the `Apple` function, you can send this variable only those messages understood by `Employees` (and not those specific to `Boss` or `VP`). These messages are safe for all `Employee` subclasses since they inherit all the behavior of their superclass.

If instead the parameter was declared as an object, not a pointer or reference:

```
void Banana(Employee e)
{
   e.print();
}
```

The compile-time type and the run-time type are both exactly `Employee`. For example, since `Employees` and `Bosses` are not necessarily the same size, it is impossible for a `Boss` object to be wedged into an `Employee`-sized space.

Compile-time binding versus run-time binding.

Go back to considering the Apple function from above. What happens when we try to send the `print` method to the parameter? If it is really pointing to an `Employee`, we expect that it will invoke `Employee's` print method, and that's fine. But what if the parameter is really pointing to a `Boss` which has its own overridden version of the `print` method?

C++ defaults to *compile-time binding*, which indicates the compiler makes the decision at compile time about which version of an overridden method to invoke. Since it is committing at compile time, it has to go with the compile-time type, so in this case it will choose to always use Employee's print method, ignoring the possibility that subclasses might provide a replacement.

This most likely is not what you intended. If `Boss` overrides print, you expect that any time you send a `Boss` a `print` message (no matter what the CT type you are working with is), it should invoke the `Boss's` version of the method. Given that OO programming was supposed to be all about making objects responsible for their own behavior, it seems uncool that it can be so easily convinced to invoke the wrong version!
What makes more sense is to use *run-time binding* where the decision about which version of the method to invoke is delayed until run-time. At the point when the `Apple` function is called, it can determine what the actual RT type is and dispatch to the correct print function for that type. This means if you call `Apple` passing an `Employee` object, it uses `Employee's print` and if you later call `Apple` passing a `Boss` object, it uses `Boss's` version of `print`.

Declaring methods `virtual`.

In C++, run-time binding is enabled on a per-method basis (although some compilers have an option to make all methods virtual, this is not standard). In order to make a particular method RT bound, you declare it virtual. Mark it `virtual` in the parent class, which makes it virtual for all subclasses, whether or not they repeat the virtual keyword on their overridden definitions. In general, you tend to want to make almost all methods `virtual` (there are a few exceptions discussed below) so that you guarantee that the right method is sent to the object without fail.

There is some performance penalty associated with RT dispatch; we'll talk about it later in the quarter, but it is not something to get too worked up over. It's most important that you are getting the correct association of method to object and without `virtual` you are leaving yourself open to problems.

Note that RT binding only applies to those objects accessed through pointers or references. If you are working with an actual object, its CT and RT types are one and the same (for example, consider the `Banana` function above) and thus there is never any difference between the CT and RT types, so it might as well bind at CT.

Constructors aren't inherited and can't be virtual.

Constructors are very tightly bound up with a class and each class has its own unique set of constructors. If `Employee` defines a 3-arg constructor, `Boss` does not inherit that constructor. If it wants to provide such a constructor, it must declare it again and just pass the arguments along to the base class constructor in the constructor initialization list. It is non-sensical to declare a constructor `virtual` since a constructor is always called by name (`Employee("Sally"...)` or `Boss("Jane"...)`) so there is no choice about which version to invoke.

Destructors aren't inherited, but should be `virtual`.

Like constructors, destructors are tightly bound with a class and each class has exactly one destructor of its own. If you don't provide a destructor, the compiler will synthesize one for you that will call the destructors of your member objects and base class and do nothing with your other data members. Note that whether you define your own or let the compiler do it for you, the destruction process will always take care of calling the superclass destructor when finished with the subclass destructor--you never explicitly invoke your parent destructor.

The destructor needs to be `virtual` for the same reason that normal methods are `virtual`, that is, you want to be sure the correct destructor is called, using the RT type of the object, not the CT type.

Consider the `Pear` function:

```
void Pear(Employee* e)
{
   delete e;
}
```

If the `Employee` destructor is not virtual, the use of delete here will be CT-bound and commit to invoking the Employee class destructor. However, if at RT the parameter was really pointing to a Boss, we really need to use the Boss's destructor to clean up the dynamically-allocated parts of a Boss object. If you declare the base class destructor as virtual, it defers the decision about which destructor to invoke until RT and then makes the correct choice. Note that declaring the destructor `virtual` makes the destructor of all subclasses virtual even though the names do not quite match `(~Employee -> ~Boss)`.
Some compilers (`gcc`, for example) will warn if you define a class with virtual methods that doesn't have a `virtual destructor`.

Operator= and copy constructor aren't inherited either.

These members are also very tightly coupled with a class. If you don't provide an operator= or copy constructor, a version is synthesized for you by the compiler. The synthesized version will do straight memberwise assignment/copying (using the assignment/copy operators of your base and member classes). If you choose to implement your own version, you should make sure to invoke the parent class to do the parent's part of the copying/assignment. For operator= that means invoking the parent version with this slightly wacky syntax:

```
Boss& Boss::operator=(const Boss& src)
{
   if (this != &src) {
      Employee::operator=(src);
      // do rest of Boss copying here
   }

   return *this;
}
```

For the copy constructor, it means chaining a call to the base class copy constructor in your constructor initialization list:

```
Boss::Boss(const Boss& src) : Employee(src)
{
   ...
}
```

See the definitions of these two methods in the Employee/Boss examples up above to see it all in context.

Assigning/copying a derived to a base "slices" the object.

If I have an object of a derived class and try to assign/copy from an object of the base class, what happens? First consider the definition of the assignment operator/copy constructor (whether explicitly defined or synthesized by the compiler). In the `Employee` class, `operator=` it will take a reference to an `Employee` object. It's completely fine to pass it a reference to a `Boss` object, since a `Boss` can always safely stand if for an `Employee`. In the copy/assign operator, it will copy the `Employee` part of the `Boss` object and ignore the extra `Boss` fields, in a sense, "slicing" out the employee fields and throwing the rest away. The result is a `Employee` object which has the same `Employee` data as the `Boss` object did, but none of the `Boss` fields or behavior is kept.

```
void Raspberry()
{
   Employee bob("Bob", .8, 10);
   Boss sally("Sally", .5, 25, "Lead Architect");

   bob = sally; // "slices" off extra fields
   bob.print(); // Bob is an *Employee* so uses Employee version
}
```

The same thing is true for the copy constructor. For example, the copy constructor is invoked when passing and returning objects by value. If I were to pass sally to the `Banana` function from above, the parameter would be a copy of just the `Employee` fields from sally. Slicing is yet another reason to avoid passing object parameters by value.

Be sure that you understand how slicing is different than the "upcast" operation where we assign a `Boss*` to a `Employee*`. In that case, we have not thrown away any information and if we're correctly using `virtual` methods, we won't lose any behavior either. Declaring a parameter as a reference/pointer to the base is simply generalizing the allowable type so that all derived types can be easily used.

Assigning/copying a base to a derived is not allowed.

In general, copying/assignment in the other direction is not allowed. The rationale goes something like this: If I were to try to assign a `Boss` from an `Employee` object, I could copy all the Employee fields, but the extra fields of a `Boss` would left uninitialized. This unsafe operation conflicts with C++'s strong commitment to ensuring all data is initialized before being used.

To be more mechanical about it, consider the definition of the assignment/copy constructor for the `Boss` class. It takes a `Boss &` as its parameter. Can I pass an `Employee&` to a function that needs a `Boss&`? Nope, an `Employee` does not necessary have all the data and methods that a `Boss` does. To make assignment work in the other direction, the `Boss` class could define a version of `operator=` that took an `Employee`, copied the `Employee` fields and do something reasonable with the remaining fields. In truth, this is not the common a need (to assign objects of different classes back and forth), but it can be done if necessary.

Calling virtual methods inside other methods.

The binding of methods called from within other methods is basically just like other bindings. For example, consider the body of the `print` method of the `Employee` class which makes a call to `askSalary()`. If `askSalary` is not declared `virtual`, the compiler binds the call at CT and within the `print` method of `Employee` "`this`" is of type `Employee*`, so it commits to using `Employee's` version. If `askSalary` is declared `virtual`, it waits until the print method is called at RT, at which point the true identity of the object is used to decide which version of `askSalary` is appropriate. It makes no difference whether the `print` method itself is declared `virtual` in deciding how to bind calls to other methods made within the `print` method.

Calling virtual functions in constructors/destructors.

The one place where `virtual` dispatch doesn't enter into the game is within constructors and destructors. If you make a call to a `virtual` function, such as `print`(), within the Employee constructor or destructor, it will always invoke the `Employee` version of print. Even if we are constructing this `Employee` object as part of the constructor of an eventual `Boss` object, at the time of the call to the `Employee` constructor, the object is actually just an `Employee` and thus responds like an `Employee`. Similarly on destruction, if a `Boss` object is being destructed, it first calls its own destructor, "stops being a Boss" and then goes on to its parent destructor. At the time of the call to `Employee` destructor, the object is no longer a `Boss`, it's just an `Employee`, and is unwinding back to its beginnings. So in a constructor/destructor the object is always of the stated CT type without exceptions. The rationale for this is that the `virtual` function may rely on part of the extra state of a `Boss` (such as the title `field`) and it will not be safe to call it before the Boss construction process has occurred, or after the `Boss` destruction has already happened.

Overriding versus overloading.

"Overloading" a method or function allows you to create functions of the same name that take different arguments. "Overriding" a method allows to replace an inherited method with a different implementation under the same name. Most often, the overridden method will have the same number and types of arguments, since it is intended to be a matching replacement. What happens when it doesn't? For example, let's say we added the "promote" method to the `Employee` class:

```
void promote(int wageIncrease);
void promote(float percentage);
```

This is method is overloaded and it chooses between the two available versions depending on whether called with a float or an int. At this point, `Boss` inherits both of these versions. Now, let's say `Boss` wants to introduce its own version of `promote`, this one taking a string which identifies a new title for the Boss:

```
void promote(char *newTitle);
```

You might like/think/hope that the `Boss` would now have all three versions of `promote`, but that isn't the way it works. In this case, the `Boss's` override of `promote` completely shadows all previous versions of `promote`, no matter what the arguments are. If we want `Boss` to have versions of `promote` that take `int` and `float`, we would need to redefine them in the `Boss` class and just provide a wrapper that calls the `inherited` version, something like this:

```
void promote(int wageIncrease) { Employee::promote(wageIncrease);}
void promote(float percentage) { Employee::promote(percentage);}
```

Seems a little awkward, but that's C++ for ya. The idea is to avoid nasty surprises where you end up getting a different inherited version when the subclass was trying to replace all of the parent's implementation of that method.

Multi-Methods

A **multi-method** is a method which is chosen according to the dynamic type of more than one object. They tend to be useful when dealing with interactions between two objects. As we have seen, `virtual` functions allow the run-time resolution of a method based on the dynamic type of an object. However, a parameter to a function can only be matched according to its `static` type. This limits us to determining which method to call to the dynamic type of one object (the object upon which the method is invoked). C++ has no built-in support for multi-methods. Other languages, such as CLOS, do have support for these. Assume we have an intersect method which tells us if two shapes intersect:

```
class Shape {
   ...
   virtual bool intersect(const Shape* s);
   ...
};

class Rectangle : public Shape {
   ...
   virtual bool intersect(const Shape* s);
   ...
};

class Circle : public Shape {
   ...
   virtual bool intersect(const Shape* s);
   ...
};
```

It doesn't make sense to see if a `Rectangle` or a `Circle` intersects a `Shape`. So we immediately see the need to provide more specialized methods:

```
class Shape {
   ...
   virtual bool intersect(const Shape *s); // Does nothing...
   virtual bool intersect(const Circle *c); // Does nothing...
   virtual bool intersect(const Rectangle *r); // Does nothing...
   ...
};

class Rectangle : public Shape {
   ...
   virtual bool intersect(const Shape *s); // Does nothing...
   virtual bool intersect(const Circle *c); // Checks circle/rectangle
   virtual bool intersect(const Rectangle *r); // Checks rectangle/rectangle
   ...
};

class Circle : public Shape {
   ...
   virtual bool intersect(const Shape* s); // Does nothing...
   virtual bool intersect(const Circle* c); // Checks circle/circle
   virtual bool intersect(const Rectangle* r); // Checks rectangle/circle
```

```
    ...
};
```

Note that we have to declare many methods which should never be called in order to prevent the hiding of methods through overloading. If we allocate a `Circle` and a `Rectangle` whose `static` types are `Shape *`, we're in for a few surprises:

```
Shape* circle = new Circle(...); // upcast to Shape
Shape* rectangle = new Rectangle(...); // upcast to Shape
circle->intersect(rectangle); // Calls Circle::intersect(Shape*)
rectangle->intersect(circle); // Calls Rectangle::intersect(Shape*)
```

We're getting one level of reification through the use of `virtual` functions, but we need two levels of reification. Short of using type fields or another similar mechanism, we need to make two virtual function calls in order to get two levels of reification. This is called **double dispatch**, and is a nice way to simulate multi-methods in C++.
We must change the methods which get called via the first virtual function call to make another virtual function call (they used to do nothing). For example we would need to write this:

```
bool Circle::intersect(Shape* shape)
{
    return shape->intersect(this); // "this" is a Circle *, not a Shape *
}
```

We would have to make a similar change to the `Rectangle::intersect(Shape *)` method.

Let's trace a call to intersect when we call it with a `Circle` and a `Rectangle`. We allocate our two shapes, both of which are bound to variables with a `static` type of `Shape *`.

```
Shape* circle = new Circle(...);       // upcast to Shape *
Shape* rectangle = new Rectangle(...); // upcast to Shape *
```

We call the intersect method, which is reified to `Circle::intersect(Shape*)`

```
circle->intersect(rectangle); // Calls Circle::intersect(Shape*)
```

The `Circle::intersect(Shape *shape)` method then executes:

```
return shape->intersect(this);
```

The dynamic type of `shape` is `Rectangle *`, and the static type of "`this`" is `Circle*`. We then call `Rectangle::intersect(Circle *)` and have thus done two levels of reification. Using double dispatch is reasonably efficient, but it requires you to write a lot of methods.