Item 2 | Polymorphism

The topic of polymorphism is given mystical status in some programming texts and is ignored in others, but it's a simple, useful concept that the C++ language supports. According to the standard, a "polymorphic type" is a class type that has a virtual function. From the design perspective, a "polymorphic object" is an object with more than one type, and a "polymorphic base class" is a base class that is designed for use by polymorphic objects.

Consider a type of financial option, `AmOption`, as shown in Figure 1.

An `AmOption` object has four types: It is simultaneously an `AmOption`, an `Option`, a `Deal`, and a `Priceable`. Because a type is a set of operations (see *Data Abstraction* [1, 1] and *Capability Queries* [27, 93]), an `AmOption` object can be manipulated through any one of its four interfaces. This means that an `AmOption` object can be manipulated by code that is written to the `Deal`, `Priceable`, and `Option` interfaces, thereby allowing the implementation of `AmOption` to leverage and reuse all that code. For a polymorphic type such as `AmOption`, the most important things inherited from its base classes are their interfaces, not their implementations. In

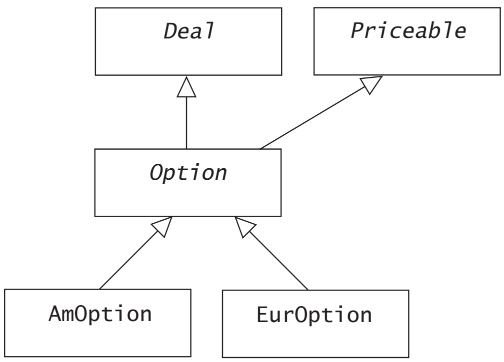

Figure 1 | Polymorphic leveraging in a financial option hierarchy. An American option has four types.

fact, it's not uncommon, and is often desirable, for a base class to consist of nothing but interface (see *Capability Queries* [27, 93]).

Of course, there's a catch. For this leveraging to work, a properly designed polymorphic class must be substitutable for each of its base classes. In other words, if generic code written to the `Option` interface gets an `AmOption` object, that object had better behave like an `Option`!

This is not to say that an `AmOption` should behave identically to an `Option`. (For one thing, it may be the case that many of the `Option` base class's operations are pure virtual functions with no implementation.) Rather, it's profitable to think of a polymorphic base class like `Option` as a contract. The base class makes certain promises to users of its interface; these include firm syntactic promises that certain member functions can be called with certain types of arguments and less easily verifiable semantic promises concerning what will actually occur when a particular member function is called. Concrete derived classes like `AmOption` and `EurOption` are subcontractors that implement the contract `Option` has established with its clients, as shown in Figure 2.

For example, if `Option` has a pure virtual `price` member function that gives the present value of the `Option`, both `AmOption` and `EurOption` must implement this function. It obviously won't implement identical behavior for these two types of `Option`, but it should calculate and return a price, not make a telephone call or print a file.

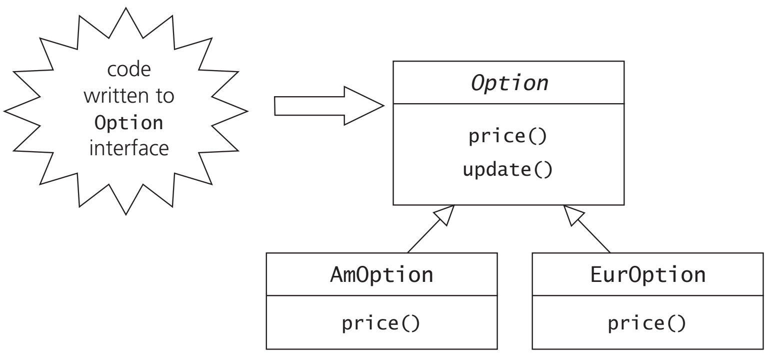

Figure 2 | A polymorphic contractor and its subcontractors. The `Option` base class specifies a contract.

On the other hand, if I were to call the `price` function of two different interfaces to the *same* object, I'd better get the same result. Essentially, either call should bind to the same function:

```
AmOption *d = new AmOption;
Option *b = d;
d->price(); // if this calls AmOption::price...
b->price(); // ...so should this!
```

This makes sense. (It's surprising how much of advanced object-oriented programming is basic common sense surrounded by impenetrable syntax.) If I were to ask you, "What's the present value of that American option?" I'd expect to receive the same answer if I'd phrased my question as, "What's the present value of that option?"

The same reasoning applies, of course, to an object's nonvirtual functions:

```
b->update(); // if this calls Option::update...
d->update(); // ...so should this!
```

The contract provided by the base class is what allows the "polymorphic" code written to the base class interface to work with specific options while promoting healthful ignorance of their existence. In other words, the polymorphic code may be manipulating `AmOption` and `EurOption` objects, but as far as it's concerned they're all just `Options`. Various concrete `Option` types can be added and removed without affecting the generic code that is aware only of the `Option` base class. If an `AsianOption` shows up at some point, the polymorphic code that knows only about `Options` will be able to manipulate it in blissful ignorance of its specific type, and if it should later disappear, it won't be missed.

By the same token, concrete option types such as `AmOption` and `EurOption` need to be aware only of the base classes whose contracts they implement and are independent of changes to the generic code. In principle, the base class can be ignorant of everything but itself. From a practical perspective, the design of its interface will take into account the requirements of its anticipated users, and it should be designed in such a way that derived classes can easily deduce and implement its contract (see *Template Method* [22, 77]). However, a base class should have no specific knowledge of any of the classes derived from it, because such knowledge inevitably makes it difficult to add or remove derived classes in the hierarchy.

In object-oriented design, as in life, ignorance is bliss (see also *Virtual Constructors and Prototype* [29, 99] and *Factory Method* [30, 103]).

Item 12 | Assignment and Initialization Are Different

Initialization and assignment are different operations, with different uses and different implementations.

Let's get it absolutely straight. Assignment occurs when you assign. All the other copying you run into is initialization, including initialization in a declaration, function return, argument passing, and catching exceptions.

Assignment and initialization are essentially different operations not only because they're used in different contexts but also because they do different things. This difference in operation is not so obvious in the built-in types such as `int` or `double`, because, in that case, both assignment and initialization consist simply of copying some bits (but see also *References Are Aliases, Not Pointers* [5, 13]):

```
int a = 12; // initialization, copy 0X000C to a
a = 12; // assignment, copy 0X000C to a
```

However, things can be quite different for user-defined types. Consider the following simple, nonstandard string class:

```
class String {
  public:
    String( const char *init ); // intentionally not explicit!
    ~String();
    String( const String &that );
    String &operator =( const String &that );
    String &operator =( const char *str );
    void swap( String &that );
    friend const String // concatenate
        operator +( const String &, const String & );
    friend bool operator <( const String &, const String & );
    //...
  private:
    String( const char *, const char * ); // computational
```

```
    char *s_;
};
```

Initializing a `String` object with a character string is straightforward. We allocate a buffer big enough to hold a copy of the character string and then copy.

```
String::String( const char *init ) {
    if( !init ) init = "";
    s_ = new char[ strlen(init)+1 ];
    strcpy( s_, init );
}
```

The destructor does what it does:

```
String::~String() { delete [] s_; }
```

Assignment is a somewhat more difficult job than construction:

```
String &String::operator =( const char *str ) {
    if( !str ) str = "";
    char *tmp = strcpy( new char[ strlen(str)+1 ], str );
    delete [] s_;
    s_ = tmp;
    return *this;
}
```

An assignment is somewhat like destruction followed by a construction. For a complex user-defined type, the target (left side, or `this`) must be cleaned up before it is reinitialized with the source (right side, or `str`). In the case of our `String` type, the `String`'s existing character buffer must be freed before a new character buffer is attached. See *Exception Safe Functions* [39, 135] for an explanation of the ordering of the statements. (By the way, just about every week somebody reinvents the bright idea of implementing assignment with an explicit destructor call and using placement new to call a constructor. It doesn't always work, and it's not exception safe. Don't do it.)

Because a proper assignment operation cleans up its left argument, one should never perform a user-defined assignment on uninitialized storage:

```
String *names = static_cast<String *>(::operator new( BUFSIZ ));
names[0] = "Sakamoto"; // oops! delete [] uninitialized pointer!
```

In this case, `names` refers to uninitialized storage because we called `operator new` directly, avoiding implicit initialization by `String`'s default constructor; `names` refers to a hunk of memory filled with random bits. When the `String` assignment operator is called in the second line, it will attempt to perform an array delete on an uninitialized pointer. (See *Placement New* [35, 119] for a safe way to perform an operation similar to such an assignment.)

Because a constructor has less work to do than an assignment operator (in that a constructor can assume it's working with uninitialized storage), an implementation will sometimes employ what's known as a "computational constructor" for efficiency:

```
const String operator +( const String &a, const String &b )
    { return String( a.s_, b.s_ ); }
```

The two-argument computational constructor is not intended to be part of the interface of the `String` class, so it's declared to be private.

```
String::String( const char *a, const char *b ) {
    s_ = new char[ strlen(a)+strlen(b)+1 ];
    strcat( strcpy( s_, a ), b );
}
```

Item 27 | Capability Queries

Most times when an object shows up for work, it's capable of performing as required, because its capabilities are advertised explicitly in its interface. In these cases, we don't ask the object if it can do the job; we just tell it to get to work:

```
class Shape {
  public:
    virtual ~Shape();
    virtual void draw() const = 0;
    //...
};
//...
Shape *s = getSomeShape(); // get a shape, and tell it to...
s->draw(); // ...get to work!
```

Even though we don't know precisely what type of shape we're dealing with, we know that it is-a `Shape` and, therefore, can `draw` itself. This is a simple and efficient—and therefore desirable—state of affairs.

However, life is not always that straightforward. Sometimes an object shows up for work whose capabilities are not obvious. For example, we may have a need for a shape that can be rolled:

```
class Rollable {
  public:
    virtual ~Rollable();
    virtual void roll() = 0;
};
```

A class like `Rollable` is often called an "interface class" because it specifies an interface only, similar to a Java interface. Typically, such a class has no non-static data members, no declared constructor, a virtual destructor, and a set of pure virtual functions that specify what a `Rollable` object is capable of doing. In this case, we're saying that anything that is-a `Rollable` can `roll`. Some shapes can roll; others can't:

```
class Circle : public Shape, public Rollable { // circles roll
    //...
    void draw() const;
    void roll();
    //...
};
class Square : public Shape { // squares don't
    //...
    void draw() const;
    //...
};
```

Of course, other types of objects in addition to shapes may be rollable:

```
class Wheel : public Rollable { ... };
```

Ideally, our code should be partitioned in such a way that we always know whether we are dealing with objects that are `Rollable` before we attempt to `roll` them, just as we earlier knew we were dealing with `Shapes` before we attempted to `draw` them.

```
vector<Rollable *> rollingStock;
//...
for( vector<Rollable *>::iterator i( rollingstock.begin() );
                                i != rollingStock.end(); ++i )
    (*i)->roll();
```

Unfortunately, we occasionally run up against situations where we simply do not know if an object has a required capability. In such cases, we are forced to perform a capability query. In C++, a capability query is typically expressed as a `dynamic_cast` between unrelated types (see *New Cast Operators* [9, 29]).

```
Shape *s = getSomeShape();
Rollable *roller = dynamic_cast<Rollable *>(s);
```

This use of `dynamic_cast` is often called a "cross-cast," because it attempts a conversion across a hierarchy, rather than up or down a hierarchy, as shown in Figure 6.

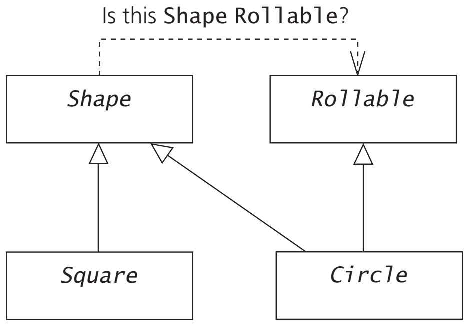

Figure 6 | A capability query: "May I tell this shape to roll?"

If `s` refers to a `Square`, the `dynamic_cast` will fail (result in a null pointer), letting us know that the `Shape` to which `s` refers is not also `Rollable`. If `s` refers to a `Circle` or to some other type of `Shape` that is also derived from `Rollable`, then the cast will succeed, and we'll know that we can `roll` the shape.

```
Shape *s = getSomeShape();
if( Rollable *roller = dynamic_cast<Rollable *>(s) )
    roller->roll();
```

Capability queries are occasionally required, but they tend to be overused. They are often an indicator of bad design, and it's best to avoid making runtime queries about an object's capabilities unless no other reasonable approach is available.

Item 55 Template Template Parameters

Let's pick up the Stack template we considered in *Specializing for Type Information* [52, 183]. We decided to implement it with a standard deque, which is a pretty good compromise choice of implementation, though in many circumstances a different container would be more efficient or appropriate. We can address this problem by adding an additional template parameter to Stack for the container type used in its implementation.

```
template <typename T, class Cont>
class Stack;
```

For simplicity, let's abandon the standard library (not usually a good idea, by the way) and assume we have available a set of nonstandard container templates: List, Vector, Deque, and perhaps others. Let's also assume these containers are similar to the standard containers but have only a single template parameter for the element type of the container.

Recall that the standard containers actually have at least two parameters: the element type and an allocator type. Containers use allocators to allocate and free their working memory so that this behavior may be customized. In effect, the allocator specifies a memory management policy for the container (see *Policies* [56, 205]). The allocator has a default so it's easy to forget it's there. However, when you instantiate a standard container like vector<int>, you're actually getting vector< int, std::allocator<int> >.

For example, the declaration of our nonstandard List would be

```
template <typename> class List;
```

Notice that we've left out the name of template parameter in the declaration of List, above. Just as with a formal argument name in a function declaration, giving a name to a template parameter in a template declaration is optional. As with a function definition, the name of a template parameter is required only in a template definition and only if the parameter name is used in the template. However, as with formal arguments in function declarations, it's common to give names to template parameters in template declarations to help document the template.

```
template <typename T, class Cont>
class Stack {
  public:
    ~Stack();
    void push( const T & );
    //...
  private:
    Cont s_;
};
```

A user of `Stack` now has to provide two template arguments, an element type and a container type, and the container has to be able to hold objects of the element type.

```
Stack<int, List<int> > aStack1; // OK
Stack<double, List<int> > aStack2; // legal, not OK
Stack<std::string, Deque<char *> > aStack3; // error!
```

The declarations of `aStack2` and `aStack3` show we have a potential problem in coordination. If the user selects the incorrect type of container for the element type, we'll get a compile-time error (in the case of `aStack3`, because of the inability to copy a `string` to a `char *`) or a subtle bug (in the case of `aStack2`, because of loss of precision in copying a `double` to an `int`). Additionally, most users of `Stack` don't want to be bothered with selection of its underlying implementation and will be satisfied with a reasonable default. We can improve the situation by providing a default for the second template parameter.

```
template <typename T, class Cont = Deque<T> >
class Stack {
    //...
};
```

This helps in cases where the user of a `Stack` is willing to accept a `Deque` implementation or doesn't particularly care about the implementation.

```
Stack<int> aStack1; // container is Deque<int>
Stack<double> aStack2; // container is Deque<double>
```

This is more or less the approach employed by the standard container adapters `stack`, `queue`, and `priority_queue`.

```
std::stack<int> stds; // container is
                      // deque< int, allocator<int> >
```

This approach is a good compromise of convenience for the casual user of the `Stack` facility and of flexibility for the experienced user to employ any (legal and effective) kind of container to hold the `Stack`'s elements.

However, this flexibility comes at a cost in safety. It's still necessary to coordinate the types of element and container in other specializations, and this requirement of coordination opens up the possibility of miscoordination.

```
Stack<int, List<int> > aStack3;
Stack<int, List<unsigned> > aStack4; // oops!
```

Let's see if we can improve safety and still have reasonable flexibility. A template can take a parameter that is itself the name of a template. These parameters have the pleasingly repetitious name of template template parameters.

```
template <typename T, template <typename> class Cont>
class Stack;
```

This new template parameter list for `Stack` looks unnerving, but it's not as bad as it appears. The first parameter, `T`, is old hat. It's just the name of a type. The second parameter, `Cont`, is a template template parameter. It's the name of a class template that has a single type name parameter. Note that we didn't give a name to the type name parameter of `Cont`, although we could have:

```
template <typename T, template <typename ElementType> class Cont>
class Stack;
```

However, such a name (`ElementType`, above) can serve only as documentation, similar to a formal argument name in a function declaration. These names are commonly omitted, but you should feel free to use them where you think they improve readability. Conversely, we could take the opportunity to reduce readability to a minimum by eliminating all technically unnecessary names in the declaration of `Stack`:

```
template <typename, template <typename> class>
class Stack;
```

But compassion for the readers of our code does impose constraints on such practices, even if the C++ language does not.

The `Stack` template uses its type name parameter to instantiate its template template parameter. The resulting container type is used to implement the `Stack`:

```
template <typename T, template <typename> class Cont>
class Stack {
    //...
  private:
    Cont<T> s_;
};
```

This approach allows coordination between element and container to be handled by the implementation of the `Stack` itself, rather than in all the various code that specializes `Stack`. This single point of specialization reduces the possibility of miscoordination between the element type and the container used to hold the elements.

```
Stack<int,List> aStack1;
Stack<std::string,Deque> aStack2;
```

For additional convenience, we can employ a default for the template template argument:

```
template <typename T, template <typename> class Cont = Deque>
class Stack {
    //...
};
//...
Stack<int> aStack1; // use default: Cont is Deque
Stack<std::string,List> aStack2; // Cont is List
```

This is often a good approach for dealing with coordination of a set of arguments to a template and a template that is to be instantiated with the arguments.

It's common to confuse template template parameters with type name parameters that just happen to be generated from templates. For example, consider the following class template declaration:

```
template <class Cont> class Wrapper1;
```

The Wrapper1 template needs a type name for its template argument. (We used the keyword class instead of typename in the declaration of the Cont parameter of Wrapper1 to tell the readers of our code that we're expecting a class or struct rather than an arbitrary type, but it's all the same to the compiler. In this context typename and class mean exactly the same thing technically. See *Optional Keywords* [63, 231].) That type name could be generated from a template, as in Wrapper1<List<int>>, but List<int> is still just a class name, even though it was generated from a template.

```
Wrapper1< List<int> > w1; // fine, List<int> is a type name
Wrapper1< std::list<int> > w2; // fine, list<int> is a type
Wrapper1<List> w3; // error! List is a template name
```

Alternatively, consider the following class template declaration:

```
template <template <typename> class Cont> class Wrapper2;
```

The Wrapper2 template needs a template name for its template argument, and not just any template name. The declaration says that the template must take a single type argument.

```
Wrapper2<List> w4; // fine, List is a template one type
Wrapper2< List<int> > w5; // error! List<int> isn't a template
Wrapper2<std::list> w6; // error! std::list takes 2+ arguments
```

If we want to have a chance at being able to specialize with a standard container, we have to do the following:

```
template <template <typename Element,
    class Allocator> class Cont>
class Wrapper3;
```

or equivalently:

```
template <template <typename,typename> class Cont>
class Wrapper3;
```

This declaration says that the template must take two type name arguments:

```
Wrapper3<std::list> w7; // might work...
Wrapper3< std::list<int> > w8; // error! list<int> is a class
Wrapper3<List> w9; // error! List takes one type argument
```

However, the standard container templates (like `list`) may legally be declared to take more than two parameters, so the declaration of `w7` above may not work on all platforms. Well, we all love and respect the STL, but we never claimed it was perfect.