Chapter 6

Polymorphism

The *polymorphic variable* is one of the most powerful mechanisms provided by the object-oriented paradigm. A polymorphic variable, you will recall, is a variable for which the static type, the type associated with a declaration, may differ from the dynamic type, the type associated with the value currently being held by the variable.

DEFINE *A static type is associated with a declaration, a dynamic type is associated with a value*

In Java a polymorphic variable can be declared as either a class type or an interface type. If a variable is declared as a class type, the value held by the variable must be either derived from the declared class or from a class that inherits from the declared class. If a variable is declared using an interface type, the value held by the variable must be derived from a class that implements the given interface. The C++ language does not include the concept of interfaces, and so the idea of a polymorphic variable is only possible using class inheritance. There are many other subtle and not-so-subtle differences between polymorphism in C++ and polymorphism in Java, as we will explain in this chapter.

To discuss polymorphism we need a class hierarchy. An intuitive hierarchy is provided by a portion of the animal kingdom, which we can represent as follows:

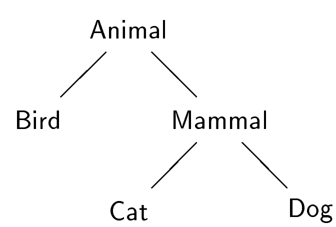


Figure 6.1 provides an realization of this class hierarchy in Java, while Figure 6.2 gives the corresponding C++ code. We will use these classes in various discussions throughout this chapter.

```
abstract class Animal {
    abstract public void speak();
}

class Bird extends Animal {
    public void speak() { System.out.println("twitter"); }
}

class Mammal extends Animal {
    public void speak() { System.out.println("can't speak"); }
    public void bark() { System.out.println("can't bark"); }
}

class Cat extends Mammal {
    public void speak() { System.out.println("meow"); }
    public void purr() { System.out.println("purrrrr"); }
}

class Dog extends Mammal {
    public void speak() { System.out.println("wouf"); }
    public void bark() { System.out.println("wouf"); }
}
```

Figure 6.1: Animal Kingdom Class Hierarchy in Java

```
class Animal {
public:
    virtual void speak() = 0;
};

class Bird : public Animal {
public:
    virtual void speak() { printf("twitter"); }
};

class Mammal : public Animal {
public:
    virtual void speak() { printf("can't speak"); }
    void bark() { printf("can't bark"); }
};

class Cat : public Mammal {
public:
    void speak() { printf("meow"); }
    virtual void purr() { printf("purrrrr"); }

};

class Dog : public Mammal {
public:
    virtual void speak() { printf("wouf"); }
    void bark() { printf("wouf"); }
};
```

Figure 6.2: Animal Kingdom Class Hierarchy in C++

6.1 Virtual and Non-virtual Overriding

DEFINE *A type signature is the type descriptions for each argument and the type description of the return type*

Overriding occurs when a method in a parent class is replaced in a child class by a method having the exact same type signature. In Java overriding is the norm, the expected result. In C++, on the other hand, whether overriding occurs or not is controlled by the programmer, using the keyword virtual.

The issue in overriding is how a method, that is, the actual code to be executed, is bound to a message. If the virtual keyword is omitted, then the binding is determined by the *static type* of a variable, that is, by the variables declared type. This is illustrated by the following:

```
Dog * d = new Dog();
Mammal * m = d;
d->bark();      // wouf
m->bark();     // can't bark
```

Because d is declared as a Dog the method selected will be that of the Dog class. Because m is declared only as a Mammal, even though it holds exactly the same value as does d, the method executed will be that provided by the class Mammal.

NOTE *Virtual overriding corresponds to the Java semantics*

If, on the other hand, a method is declared as *virtual*, as is the speak method in Figure 6.2, then the method invoked may, under the right circumstances, be determined by the *dynamic* (that is, run time) value held by the class. This is illustrated by the following:

```
d->speak();     // wouf
m->speak();     // also wouf
Animal * a = d;
a->speak();     // and more wouf
```

The method speak for variable d will be that of class Dog, as might be expected. However m will also use the Dog method. Even a, which is an Animal, will use the Dog method. Thus, virtual overridding corresponds to the behavior of overridden functions in Java.

Regardless of the type of value held by a variable, the validity of a message is determined by the static, or declared type. This is the same as in Java. Thus while the variable d will respond to the message bark, the variable a that was declared as Animal, even though it contains the exact same value, is not allowed to perform this operation.

```
d->bark();     // wouf
a->bark();     // compile error, not allowed
```

Because of the C++ memory model (see Chapter 4) virtual, or polymorphic, overriding will only occur when used with a pointer or reference, and not with stack-based variables. This is illustrated by the following:

```
Mammal mm;
mm = *d;
mm.speak();     // can't speak
d->speak();     // although dog will wouf
```

Note that the variable mm is here not declared as a pointer, as were earlier variables, but as a simple stack-based value. The Dog value held by d is assigned to the variable mm. During the assignment process the value loses its dog identity, and becomes simply a mammal. Thus the speak method will be that of class Mammal, not that of class Dog.

The pseudo-variable this, the reference to the receiver within a method, is a pointer in C++, whereas it is an object value in Java. Thus, implicit messages sent to this can have polymorphic bindings.

NOTE *The variable* this *is a pointer in C++, a variable in Java*

If a variable is not declared as virtual in a parent class, it cannot subsequently be made virtual in a child class. On the other hand, the keyword is optional in the child class; once declared as virtual by the parent, the method remains virtual in all child class definitions. Notice that we have made use of this fact by omitting the virtual keyword from the specification of the method speak in class Cat. Despite this, the method still remains virtual.

6.1.1 Impact of Virtual on Size

When a class description contains a virtual method, an internal table is created for the class, called the virtual method table. This table is used in the implementation of the dynamic binding of message to method required by the virtual method. In order to do this, each instance of a class will contain an additional hidden pointer value, which references the virtual method table. The programmer can see this effect by adding or removing the virtual keyword from a class description, and examining the size of an instance of the class:

NOTE *When classes contain virtual methods, instances will hold a hidden pointer to a virtual method table*

```
class sizeTest {
    public:
        int x, y;
        virtual void test () { x = 3; }
};

sizeTest x;
    // size will be 8 if not virtual, larger if virtual
println("%d\n", sizeof(x) );
```

6.1.2 Obtaining Type Information from a dynamic value

In Java all objects recognize the method getClass(), and in response will yield a Class object that describes the dynamic type of the value. Using the Class value one can obtain various bits of information about the value, for example a string that describes the dynamic type:

```
Animal a = new Dog();
    // following will print class Dog
System.out.println("class is " + a.getClass().getName());
```

WARNING *The* typeid *feature is a recent addition to C++*

The equivalent feature in C++ is a function named typeid. The typeid function returns a value of type typeinfo, described by the include file of the same name. The string representation of the name of the class if yielded by the method name:

```
Animal * a = new Dog();
    // will print the class Dog
println("class is %s\n", typeid(*a).name());
```

Notice it is necessary to dereference the variable a, since the typeid must act on the value a points to, not the pointer value itself.

Being a relatively recent addition to C++, the typeid facility is one of the few places in the standard library that will generate an exception on error. Should the pointer value in the above expression be null, a bad_typeid exception will be thrown.

6.2 Abstract Classes

DEFINE *A pure virtual method must be overridden in subclasses*

An *abstract class* is a class used only as a parent class for inheritance, one that cannot be used to create instances directly. The Java language includes an explicit keyword abstract to indicate this situation. The C++ language does not use this keyword. Instead, an abstract class is simply a class that includes a *pure virtual method*. A pure virtual method is a method that is declared as virtual but which does not include a method body. Instead, the method is "assigned" the null value. An example occurs in Figure 6.2:

```
class Animal {
public:
    virtual void speak() = 0;
};
```

As with abstract classes in Java, it is not possible to create an instance of a class that contains a pure virtual member. An attempt to do so will produce a compile time error message.

As we noted at the beginning of this chapter, the C++ language does not provide the interface facility. Sometimes classes that consist entirely of pure virtual methods are used in the same manner as interfaces:

NOTE *An interface can be simulated by pure virtual methods*

```
class KeyPressHandler {     // specification for key press event handler
public:
    virtual void keyDown (char c) = 0;
};

class MouseDownHandler { // specification for mouse down event handler
public:
    virtual void mouseDown (int x, int y) = 0;
    virtual void mouseUp (int x, int y) = 0;
};
```

Since C++ supports multiple inheritance (see Section 12.8), a class can implement several of such interfaces:

```
class EventHandler : public KeyPressHandler, public MouseDownHandler {
public:
    void keyDown (char c) { ... }
    void mouseDown (int x, int y) { ... }
    void mouseUp (int x, int y)( { ... }
};
```

In Java the keyword final is in some ways the opposite of abstract, serving to indicate methods or classes that cannot be overwritten. There is no equivalent feature in C++, although as we noted in the previous chapter in some cases declaring the constructor for a class as protected can have a similar effect.

6.3 Downcasting (Reverse Polymorphism)

A polymorphic variable can have a dynamic type that is a subclass of its static, or declared type. For example, a variable can be declared as a pointer to an Animal, but actually be maintaining a pointer to a Cat. Often one is required to form an assignment that depends upon the dynamic type, rather than the static type. For example, one needs to assign the polymorphic Animal variable to a variable of type Cat.

NOTE *Downcasting reverses the assignment to a polymorphic variable, hence the term reverse polymorphism*

The Java programmer can test the dynamic type of a variable by means of the operator instanceof, and will perform the transformation by using a cast operator:

```
Animal a = ... ;
```

```
if (a instanceof Cat)
    Cat c = (Cat) a;
```

Alternatively, the Java programmer can explicitly catch the exception that is thrown if the conversion is illegal:

```
Animal a = ...;
try {
    Cat c = (Cat) a;
} catch(ClassCastException & e) { ... }
```

There is no direct C++ equivalent to the instanceof operation. Furthermore, although the syntax of the cast operation is taken directly from C++, the Java programmer should be aware that the semantics of the equivalent operation in C++ are slightly different. The Java cast performs a run-time check to ensure the validity of the conversion, and issues an exception if illegal. The C++ cast is entirely a compile time operation, and no check is made at run-time. If the cast is improper no indication is given to the programmer, and an erroneous outcome will likely result:

WARNING *C++ does not perform a run time check to ensure the validity of cast conversions*

```
Animal * a = new Dog();
Cat * c = (Cat *) a;
c->purr();      // behavior is undefined
```

Such errors can sometimes be hidden due to the interaction between the cast operation and the rules for virtual and nonvirtual method invocation. Note, for example, that if we had *not* declared the method purr as virtual, the proper Cat method would have been invoked (since the static type is Cat) despite the fact that the actual value held by variable c is a dog. The behavior when the method *is* declared as virtual is more difficult to predict; on many machines it will produce a segmentation fault.

NOTE *The RTTI is a recent addition to the C++ language*

To get around this problem the C++ language provides a different type of cast, called a *dynamic cast*. The dynamic cast is part of a suite of functions, called the run-time type information system, or RTTI. The dynamic cast operator is a templated function (see Chapter 9). The template argument is the type to which conversion is desired. Unlike the normal cast, the dynamic cast operator checks the validity of the conversion. If the conversion is not proper, a null value is yielded. Thus, the result is either a properly type-checked value, or null. The programmer can then test the resulting value to see if the conversion took place. In this manner the dynamic_cast operator combines the features of both the instanceof operator and the cast operator in Java.

NOTE *Testing a pointer in an* if *statement is the same as testing whether or not the pointer is null*

```
Cat * c = dynamic_cast<Cat *>(a);
if (c)
    printf("variable was a cat");
```

```
else
    printf("variable was not a cat");
```

If dynamic_cast is used with object values, instead of pointers, a failure results in a bad_cast exception being thrown, rather than a null pointer. The dynamic cast operation works only with polymorphic types, that is, pointers (or references) to classes that contain at least one virtual method.

A static_cast is similar, but performs no dynamic check on the result. This is most often used to convert one pointer type, for example a void * pointer, into another type:

```
void * v = ...;

    // we know, from elsewhere, that v is really a cat
Cat * c = static_cast<Cat *>(v);
```

A static cast is not restricted to polymorphic types. Two other types of cast (const_cast, and reinterpret_cast) have also been added to C++, but their use is uncommon and they will not be described here. However, programmers are encouraged to use these newer, more type-safe facilities instead of the older cast mechanism.

RULE
Whenever possible, use the RTTI instead of standard unchecked cast conversions

6.3.1 Simulating The Dynamic Cast

The RTTI is a relatively new addition to the C++ language, and not all compilers will yet support this feature. Thus, it may be necessary to achieve the effect of the dynamic cast operator without actually using the operator. Before the introduction of RTTI, one common programmers trick was to encode explicit *is-a* methods in class hierarchies. For example, to test animal values to see if they represent a dog or cat, we can write methods such as the following:

```
class Mammal {
public:
    virtual bool  isaDog() { return false; }
    virtual bool  isaCat() { return false; }
};

class Dog : public Mammal {
public:
    virtual bool  isaDog() { return true; }
};

class Cat : public Mammal {
public:
```

```
    virtual bool  isaCat() { return true; }
};

Mammal * fido;
```

A test, such as fido→isaDog(), can then be used to determine if the variable fido is currently holding a value of type Dog. If so, a conventional cast can safely be used to convert the quantity into the correct type.

By returning a pointer rather than an integer, we can extend this trick to combine both the test for subclass type and the conversion, which is more closely similar to the dynamic cast operator in the RTTI. Since a function in the class Mammal is returning a pointer to a Dog, the class Dog must have a forward reference (see Section 5.3). The result of the assignment is either a null pointer or a valid reference to a Dog; so, the test on the result must still be performed but we have eliminated the need for the cast. This is shown as follows:

```
class Dog;    // forward reference
class Cat;

class Mammal {
public:
    virtual Dog *  isaDog() { return 0; }
    virtual Cat *  isaCat() { return 0; }
};

class Dog : public Mammal {
public:
    virtual Dog *  isaDog() { return this; }
};

class Cat : public Mammal {
public:
    virtual Cat *  isaCat() { return this; }
};

Mammal * fido;
Dog * lassie;
```

A statement such as

```
lassie = fido->isaDog();
```

can then *always* be performed. It will result in the variable lassie holding a non-null value only if fido indeed held a value of class Dog. If fido did *not* hold a dog value, then a null pointer value will be assigned to the variable lassie.

```
if (lassie)
    ... fido was indeed a dog
else
    ... assignment did not work
    ... fido was not a dog
```

While it is possible for the programmer to implement this, the disadvantage of this technique for performing downcasting (sometimes called reverse polymorphism) is that it requires adding methods to both the parent and the child classes. If there are many child classes inheriting from one common parent class, the mechanism can become unwieldy. If making changes to the parent class is not permitted this technique is not possible.

6.4 Name Resolution

As part of object oriented method invocation a message selector must be bound to the appropriate function body. The techniques used by Java and C++ for this purpose are similar, but not identical. Consider, for example, the following two class definitions in Java:

DEFINE *Name resolution is matching a function body to a function name*

```
class Parent {
    public void test (int i)
        { System.out.println("parent test"); }
}

class Child extends Parent {
    public void test (int i, int i) {
        System.out.println("child two arg test"); }
    public void test (Parent p) {
        System.out.println("child object test"); }
}
```

The name space for the class Parent introduces a new function, named test, that takes a single integer argument. The class Child builds on this name space, and adds to this two other definitions for the function test. Each of these can be easily distinguished from the original by the number or type of arguments, so there is no possibility of confusion. If we now provide an invocation, such as the following:

```
Child c = new Child();
```

```
c.test(3);
```

the compiler selects the function with matching arguments, in this case the function inherited from the class Parent.

Now consider an equivalent C++ program:

```
class Parent {
public:
    void test (int i) { printf("parent test"); }
};

class Child : public Parent {
public:
    void test (int i, int i) { printf("child two arg test"); }
    void test (Parent & p) { printf("child object test"); }
};
```

If we try invoking the function inherited from the parent, we will get a compiler error:

```
Child * c = new  Child();
c->test(3);    // will generate compiler error
```

The explanation for this behavior is that the C++ language maintains separate but linked descriptions of each of the various name scopes. In this case, there are at least three different name scopes: the global scope, the scope for class Parent, and the scope for class Child. To resolve a name, such as test, the compiler performs a two step process. Step one is to search for the first enclosing scope in which the name is defined. In this case, that would be the scope for Child. Step two is to then try to match the name with a function *defined in that scope*. In this case, there are only two possibilities, neither of which will work. Being unable to find a matching function, a compiler error is reported.

RULE *Redefine any inherited names that are overloaded with different type signatures*

To circumvent this, the C++ programmer should redefine any inherited names that are being overloaded with new meanings. This can be done with a simple in-line function, as in the following:

```
class Child : public Parent {
public:
    void test (int i) { Parent::test(i); } // redefine inherited method
    void test (int i, int i) { printf("child two arg test"); }
    void test (Parent & p) { printf("child object test"); }
};
```

Now all three methods will be defined in the Child scope, and will hence be available for use.

6.5 A Forest, not a Tree

In Java all objects descend ultimately from the base class Object. This has the advantage of ensuring that every object possesses some minimal functionality, namely the methods provided by class Object. These operations include the ability to get the class of an object, convert an object into a string representation, test an object for equality against another object, and compute the hash value for an object.

Classes in C++ are not part of a single hierarchy. If a class is not defined as inheriting from another class, then it is the root of its own hierarchy, and provides only the behavior defined by the class description. Thus, a typical C++ program contains a number of different class hierarchies, each independent of the others.

NOTE *In C++ there is no class that is ancestor to all classes*

In Java the class Object is often used to declare universal generic objects, values that can hold any other object type. Since C++ does not have a single root class, there is no exact equivalence. Frequently template classes (see Chapter 9) eliminate the need for generic Object variables. However, where they cannot be avoided, void pointers can often be made to serve the same purpose. A variable declared as a pointer to a void value can be assigned any other pointer type, regardless of the type of object the pointer references.

```
Animal * a = new Dog();
void * v = a;     // assign v pointer to an animal
```

Just as a cast must be used to downcast an Object value in Java, a dynamic cast (see Section 6.3) should be used to convert a void pointer value back into the original type.

```
Dog * dd = dynamic_cast<Dog *>(v);
```

Note, however, that the dynamic cast only works if the pointer references a class that contains at least one virtual method.

6.6 Virtual Destructors

A *destructor* (see Chapter 4) is a method that is invoked immediately before a variable is to be deleted. When polymorphic variables are used, a concern is whether or not a destructor function should be declared as virtual. To illustrate, let us add destructor functions to the classes presented earlier in Figure 6.2:

```
class Animal {
```

```
    virtual ~Animal () { printf("goodbye animal"); }
    ...
};

...

class Cat : public Mammal {
    ~Cat () { printf("goodbye cat"); }
    ...
};
```

Now imagine we create and delete a polymorphic variable, as follows:

```
Animal * a = new Cat();
delete a;
```

If the destructor in Animal is declared virtual, as shown, then both the destructors in class Animal and class Cat will be executed. If the virtual designation is omitted, then only the method in class Animal will be performed. If the destructor is omitted from Animal altogether, then the method from class Cat will not be performed, whether or not it is declared virtual.

RULE *Declare a virtual destructor if a class has any virtual methods*

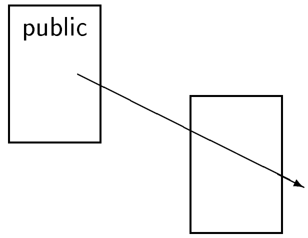
A good rule of thumb is to declare a destructor as virtual if there are any other virtual methods. A destructor should be provided in this case, even if it performs no useful actions, as otherwise destructors from child classes may not be executed.

Note also one more difference between destructors and finalize methods in Java. A finalize method should always explicitly invoke the finalize method that it inherits from its parent class. A destructor will do this automatically, and no explicit call is required.

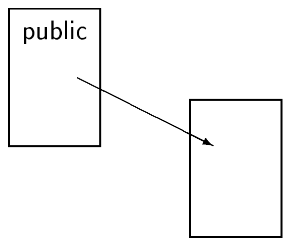
6.7 Private Inheritance

You will have undoubtedly noticed how the keyword public is used to indicate inheritance in C++, rather than the keyword extends as in Java. While public inheritance is the most common, it is also possible to perform protected or private inheritance. When one of these other forms are used, the visibility of data fields and methods is the maximum of their declared modifiers and the modifier used for inheritance. That is, if inheritance is protected, then fields declared as public in the parent class become protected in the child class. If inheritance is private, then fields declared either as public or protected in the parent become private in the child class.

To understand the significance of this distinction, imagine the public features of a parent class as flowing through a child class, to become public features of the child class as well:



In a private inheritance, the public (and protected) features of the parent class are available for use in the child class, but do not become part of the child class interface. In effect, they do not flow through the child class, but are instead stopped at that level:



To illustrate why you might want to use this feature, imagine that you need to build a stack abstraction, and you have already a list class that you want to use as the underlying container. One possibility is to simply use inheritance, and derive the stack from the list:

```
class Stack : public List { // assume elements are integers
public:
    push (int val) { addToFront(val); }
    pop () { removeFirstElement(); }
    top () { return firstElement();
}
```

A problem with this abstraction is that it is too powerful, it provides the user of the stack with too many operations. In particular, there is no way to keep the user from accessing the List operations, even when they are not appropriate. For example, somebody might directly add or remove and element directly from the bottom of a stack.

By specifying a private inheritance, we avoid this potential misuse. The features of the parent class List, even if they are declared public or protected, are not passed through to become part of the Stack interface. Thus, the only features are those explicitly described:

RULE *Use private inheritance when the child class is not a more specialized form of the parent class*

```
class Stack : private List {
public:
    push (int val) { addToFront(val); }
```

```
    pop () { removeFirstElement(); }
    top () { return firstElement();
}
```

But if public inheritance permitted too many operations to become attached to the new abstraction, simply declaring a private inheritance can be too restrictive. There may be some operations that one wants to permit. For example, the methods that check the size of the list are still appropriate for the stack abstraction. We can specify that these new features should continue to be part of the Stack abstraction by means of the using keyword. The using keyword permits individual items from the parent class to be selected and attached to the interface for the child class, while filtering out all other operations.

```
class Stack : private List {
public:
    push (int val) { addToFront(val); }
    pop () { removeFirstElement(); }
    top () { return firstElement(); }
    using isEmpty();
    using int size();
};
```

6.8 Inheritance and Arrays

There are a number of situations where it could be argued that the Java semantics are an improvement over the C++ semantics, most often because the C++ semantics are incomplete or undefined. However, there is one curious situation where the Java semantics seem more confused than their C++ counterpart. This concerns an interaction between inheritance and arrays. Assume we have declared an array of Dog values. Java permits this array to be assigned to a variable that is declared as an array of the parent class:

```
Dog [ ] dogs = new Dog[10]; // an array of dog values
Animal [ ] pets = dogs; // legal
```

In effect, Java is asserting that the type Dog[ ] (that is, array of dogs) is a subtype of the type Animal[ ]. To see what confusion can then arize, imagine the following assignment:

```
pets[2] = aCat; // is this legal?
```

On the face of it, it would seem to certainly be legal to reassign an element in the array to now hold a Cat value. After all, the array is declared as an array of animals, and a Cat is an animal. But remember that the array in question shares a reference with an array of dog values, and by performing this assignment we actually convert one element in the Dog array into a cat.

To prevent this, Java actually performs a run-time check on assignments to arrays of objects. C++, on the other hand, takes a simpler approach, and simply asserts that even though a Dog may be an Animal, there is no inheritance or subtype relationship between an array of Dog and an array of Animal.

6.9 Overloading

A function is said to be *overloaded* when there are two or more function *bodies* associated with a single function *name*. Overriding is one form of overloading, however overloading can occur even without overriding. We saw an example of this in an earlier section, which included the following class definition:

DEFINE
An overloaded name has more than one meaning

```
class Child : public Parent {
public:
    void test (int i) { Parent::test(i); } // redefine inherited method
    void test (int i, int j) { printf("child two arg test"); }
    void test (Parent & p) { printf("child object test"); }
};
```

Here there are three different versions of the test function, distinguished by the compiler by the number and type of arguments used in the function invocation. Constructor functions are often overloaded in this fashion, however any function can be so defined.

The Java programmer should be aware that almost all C++ operators can also be overloaded. For example, if we wanted to provide a meaning for the operations of "adding" two cats or two dogs, we could do so as follows:

```
Dog * operator + (Dog * left, Dog * right)
{
    // return a new Dog value
    // that is the sum of the parents
    return new Dog();
}

Cat * operator + (Cat * left, Cat * right)
{
    return new Cat();
}
```

These functions would permit a dog value to be added to another dog, or a cat to a cat, but not permit a cat to be added to a dog. Operators can be defined either as ordinary functions (as shown here) or as member functions. This will be discussed in detail in the next chapter.

Test Your Understanding

1. What is a polymorphic variable?
2. Using the concepts of static and dynamic type, explain the effect of the modifier `virtual`.
3. How can you print the name of the class for an object value being held by a polymorphic variable?
4. What is a pure virtual method?
5. What is a downcast?
6. What do the initial RTTI stand for?
7. What is a dynamic cast? How does it differ from a normal cast?
8. Explain how the name resolution algorithm used in C++ differs from that of Java.
9. How are exceptions tied to function names in C++? How is this different from Java?
10. What are some of the advantages Java derives from having all object types inherit from the same base class (namely, `Object`)?
11. What is a virtual destructor? When is such a concept important?
12. How does private inheritance differ from normal inheritance?
13. What is an overloaded name? How is it different from an overridden method name?