Thinking in C++, 2nd ed. Volume 1

©2000 by Bruce Eckel

15: Polymorphism & Virtual Functions

Polymorphism (implemented in C++ with **virtual** functions) is the third essential feature of an object-oriented programming language, after data abstraction and inheritance.

It provides another dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism allows improved code organization and readability as well as the creation of *extensible* programs that can be "grown" not only during the original creation of the project, but also when new features are desired.

Encapsulation creates new data types by combining characteristics and behaviors. Access control separates the interface from the implementation by making the details **private**. This kind of mechanical organization makes ready sense to someone with a procedural programming background. But virtual functions deal with decoupling in terms of *types*. In Chapter 14, you saw how inheritance allows the treatment of an object as its own type *or* its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally. The virtual function allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the functions that you can call through the base class.

In this chapter, you'll learn about virtual functions, starting from the basics with simple examples that strip away everything but the "virtualness" of the program.

Evolution of C++ programmers

C programmers seem to acquire C++ in three steps. First, as simply a "better C," because C++ forces you to declare all functions before using them and is much pickier about how variables are used. You can often find the errors in a C program simply by compiling it with a C++ compiler.

The second step is "object-based" C++. This means that you easily see the code organization benefits of grouping a data structure together with the functions that act upon it, the value of constructors and destructors, and perhaps some simple inheritance. Most programmers who have been working with C for a while quickly see the usefulness of this because, whenever they create a library, this is exactly what they try to do. With C++, you have the aid of the compiler.

You can get stuck at the object-based level because you can quickly get there and you get a lot of benefit without much mental effort. It's also easy to feel like you're creating data types – you make classes and objects, you send messages to those objects, and everything is nice and neat.

But don't be fooled. If you stop here, you're missing out on the greatest part of the language, which is the jump to true object-oriented programming. You can do this only with virtual functions.

Virtual functions enhance the concept of type instead of just encapsulating code inside structures and behind walls, so they are without a doubt the most difficult concept for the new C++ programmer to fathom. However, they're also the turning point in the understanding of object-oriented programming. If you don't use virtual functions, you don’t understand OOP yet.

Because the virtual function is intimately bound with the concept of type, and type is at the core of object-oriented programming, there is no analog to the virtual function in a traditional procedural language. As a procedural programmer, you have no referent with which to think about virtual functions, as you do with almost every other feature in the language. Features in a procedural language can be understood on an algorithmic level, but virtual functions can be understood only from a design viewpoint.

Upcasting

In Chapter 14 you saw how an object can be used as its own type or as an object of its base type. In addition, it can be manipulated through an address of the base type. Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called *upcasting* because of the way inheritance trees are drawn with the base class at the top.

You also saw a problem arise, which is embodied in the following code:

```
//: C15:Instrument2.cpp
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

The function **tune( )** accepts (by reference) an **Instrument**, but also without complaint anything derived from **Instrument**. In **main( )**, you can see this happening as a **Wind** object is passed to **tune( )**, with no cast necessary. This is acceptable; the interface in **Instrument** must exist in **Wind**, because **Wind** is publicly inherited from **Instrument**. Upcasting from **Wind** to **Instrument** may "narrow" that interface, but never less than the full interface to **Instrument**.

The same arguments are true when dealing with pointers; the only difference is that the user must explicitly take the addresses of objects as they are passed into the function.

The problem

The problem with **Instrument2.cpp** can be seen by running the program. The output is **Instrument::play**. This is clearly not the desired output, because you happen to know that the object is actually a **Wind** and not just an **Instrument**. The call should produce **Wind::play**. For that matter, any object of a class derived from **Instrument** should have its version of **play( )** used, regardless of the situation.

The behavior of **Instrument2.cpp** is not surprising, given C's approach to functions. To understand the issues, you need to be aware of the concept of *binding*.

Function call binding

Connecting a function call to a function body is called *binding*. When binding is performed before the program is run (by the compiler and linker), it's called *early binding*. You may not have heard the term before because it's never been an option with procedural languages: C compilers have only one kind of function call, and that's early binding.

The problem in the program above is caused by early binding because the compiler cannot know the correct function to call when it has only an **Instrument** address.

The solution is called *late binding*, which means the binding occurs at runtime, based on the type of the object. Late binding is also called *dynamic binding* or *runtime binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at runtime and call the appropriate member function. In the case of a compiled language, the compiler still doesn't know the actual object type, but it inserts code that finds out and calls the correct function body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects. You'll see how this works later.

virtual functions

To cause late binding to occur for a particular function, C++ requires that you use the **virtual** keyword when declaring the function in the base class. Late binding occurs only with **virtual** functions, and only when you're using an address of the base class where those **virtual** functions exist, although they may also be defined in an earlier base class.

To create a member function as **virtual**, you simply precede the declaration of the function with the keyword **virtual**. Only the declaration needs the **virtual** keyword, not the definition. If a function is declared as **virtual** in the base class, it is **virtual** in all the derived classes. The redefinition of a **virtual** function in a derived class is usually called *overriding*.

Notice that you are only required to declare a function **virtual** in the base class. All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism. You *can* use the **virtual** keyword in the derived-class declarations (it does no harm to do so), but it is redundant and can be confusing.

To get the desired behavior from **Instrument2.cpp**, simply add the **virtual** keyword in the base class before **play( )**:

```
//: C15:Instrument3.cpp
// Late binding with the virtual keyword
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Override interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

This file is identical to **Instrument2.cpp** except for the addition of the **virtual** keyword, and yet the behavior is significantly different: Now the output is **Wind::play**.

Extensibility

With **play( )** defined as **virtual** in the base class, you can add as many new types as you want without changing the **tune( )** function. In a well-designed OOP program, most or all of your functions will follow the model of **tune( )** and communicate only with the base-class interface. Such a program is *extensible* because you can add new functionality by inheriting new data types from the common base class. The functions that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Here's the instrument example with more virtual functions and a number of new classes, all of which work correctly with the old, unchanged **tune( )** function:

```
//: C15:Instrument4.cpp
// Extensibility in OOP
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
```

```
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
```

```
    Woodwind recorder;
    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);
    f(flugelhorn);
  } ///:~
```

You can see that another inheritance level has been added beneath **Wind**, but the **virtual** mechanism works correctly no matter how many levels there are. The **adjust( )** function is *not* overridden for **Brass** and **Woodwind**. When this happens, the "closest" definition in the inheritance hierarchy is automatically used – the compiler guarantees there's always *some* definition for a virtual function, so you'll never end up with a call that doesn't bind to a function body. (That would be disastrous.)

The array **A[ ]** contains pointers to the base class **Instrument**, so upcasting occurs during the process of array initialization. This array and the function **f( )** will be used in later discussions.

In the call to **tune( )**, upcasting is performed on each different type of object, yet the desired behavior always takes place. This can be described as "sending a message to an object and letting the object worry about what to do with it." The **virtual** function is the lens to use when you're trying to analyze a project: Where should the base classes occur, and how might you want to extend the program? However, even if you don't discover the proper base class interfaces and virtual functions at the initial creation of the program, you'll often discover them later, even much later, when you set out to extend or otherwise maintain the program. This is not an analysis or design error; it simply means you didn't or couldn't know all the information the first time. Because of the tight class modularization in C++, it isn't a large problem when this occurs because changes you make in one part of a system tend not to propagate to other parts of the system as they do in C.

How C++ implements late binding

How can late binding happen? All the work goes on behind the scenes by the compiler, which installs the necessary late-binding mechanism when you ask it to (you ask by creating virtual functions). Because programmers often benefit from understanding the mechanism of virtual functions in C++, this section will elaborate on the way the compiler implements this mechanism.

The keyword **virtual** tells the compiler it should not perform early binding. Instead, it should automatically install all the mechanisms necessary to perform late binding. This means that if you call **play( )** for a **Brass** object *through an address for the base-class* **Instrument**, you'll get the proper function.

To accomplish this, the typical compiler[54] creates a single table (called the VTABLE) for each class that contains **virtual** functions. The compiler places the addresses of the virtual functions for that particular class in the VTABLE. In each class with virtual functions, it secretly places a pointer, called the *vpointer* (abbreviated as VPTR), which points to the VTABLE for that object. When you make a virtual function call through a base-class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the correct function and causing late binding to take place.

All of this – setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call – happens automatically, so you don't have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

The following sections go into this process in more detail.

Storing type information

You can see that there is no explicit type information stored in any of the classes. But the previous examples, and simple logic, tell you that there must be some sort of type information stored in the objects; otherwise the type could not be established at runtime. This is true, but the type information is hidden. To see it, here's an example to examine the sizes of classes that use virtual functions compared with those that don't:

```
//: C15:Sizes.cpp
```

```
// Object sizes with/without virtual functions
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
```

```
    cout << "OneVirtual: "
         << sizeof(OneVirtual) << endl;
    cout << "TwoVirtuals: "
         << sizeof(TwoVirtuals) << endl;
  } ///:~
```

With no virtual functions, the size of the object is exactly what you'd expect: the size of a single[55] **int**. With a single virtual function in **OneVirtual**, the size of the object is the size of **NoVirtual** plus the size of a **void** pointer. It turns out that the compiler inserts a single pointer (the VPTR) into the structure if you have one *or more* virtual functions. There is no size difference between **OneVirtual** and **TwoVirtuals**. That's because the VPTR points to a table of function addresses. You need only one table because all the virtual function addresses are contained in that single table.

This example required at least one data member. If there had been no data members, the C++ compiler would have forced the objects to be a nonzero size because each object must have a distinct address. If you imagine indexing into an array of zero-sized objects, you'll understand. A "dummy" member is inserted into objects that would otherwise be zero-sized. When the type information is inserted because of the **virtual** keyword, this takes the place of the "dummy" member. Try commenting out the **int a** in all the classes in the example above to see this.

Picturing virtual functions

To understand exactly what's going on when you use a virtual function, it's helpful to visualize the activities going on behind the curtain. Here's a drawing of the array of pointers **A[ ]** in **Instrument4.cpp**:

The array of **Instrument** pointers has no specific type information; they each point to an object of type **Instrument**. **Wind**, **Percussion**, **Stringed**, and **Brass** all fit into this category because they are derived from **Instrument** (and thus have the same interface as **Instrument**, and can respond to the same messages), so their addresses can also be placed into the array. However, the compiler doesn't know that they are anything more than **Instrument** objects, so left to its own devices it would normally call the base-class versions of all the functions. But in this case, all those functions have been declared with the **virtual** keyword, so something different happens.

Each time you create a class that contains virtual functions, or you derive from a class that contains virtual functions, the compiler creates a unique VTABLE for that class, seen on the right of the diagram. In that table it places the addresses of all the functions that are declared virtual in this class or in the base class. If you don't override a function that was declared virtual in the base class, the compiler uses the address of the base-class version in the derived class. (You can see this in the **adjust** entry in the **Brass** VTABLE.) Then it places the VPTR (discovered in **Sizes.cpp**) into the class. There is only one VPTR for each object when using simple inheritance like this. The VPTR must be initialized to point to the starting address of the appropriate VTABLE. (This happens in the constructor, which you'll see later in more detail.)

Once the VPTR is initialized to the proper VTABLE, the object in effect "knows" what type it is. But this self-knowledge is worthless unless it is used at the point a virtual function is called.

When you call a virtual function through a base class address (the situation when the compiler doesn't have all the information necessary to perform early binding), something special happens. Instead of performing a typical function call, which is simply an assembly-language **CALL** to a particular address, the compiler generates different code to perform the function call. Here's what a call to **adjust( )** for a **Brass** object looks like, if made through an **Instrument** pointer (An **Instrument** reference produces the same result):

The compiler begins with the **Instrument** pointer, which points to the starting address of the object. All **Instrument** objects or objects derived from **Instrument** have their VPTR in the same place (often at the beginning of the object), so the compiler can pick the VPTR out of the object. The VPTR points to the starting address of the VTABLE. All the VTABLE function addresses are laid out in the same order, regardless of the specific type of the object. **play( )** is first, **what( )** is second, and **adjust( )** is third. The compiler knows that regardless of the specific object type, the **adjust( )** function is at the location VPTR+2. Thus, instead of saying, "Call the function at the absolute location **Instrument::adjust**" (early binding; the wrong action), it generates code that says, in effect, “Call the function at VPTR+2.” Because the fetching of the VPTR and the determination of the actual function address occur at runtime, you get the desired late binding. You send a message to the object, and the object figures out what to do with it.

Under the hood

It can be helpful to see the assembly-language code generated by a virtual function call, so you can see that late-binding is indeed taking place. Here’s the output from one compiler for the call

```
    i.adjust(1);
```

inside the function **f(Instrument& i)**:

```
    push  1
    push  si
    mov   bx, word ptr [si]
    call  word ptr [bx+4]
    add   sp, 4
```

The arguments of a C++ function call, like a C function call, are pushed on the stack from right to left (this order is required to support C’s variable argument lists), so the argument **1** is pushed on the stack first. At this point in the function, the register **si** (part of the Intel X86 processor architecture) contains the address of **i**. This is also pushed on the stack because it is the starting address of the object of interest. Remember that the starting address corresponds to the value of **this**, and **this** is quietly pushed on the stack as an argument before every member function call, so the member function knows which particular object it is working on. So you’ll always see one more than the number of arguments pushed on the stack before a member function call (except for **static** member functions, which have no **this**).

Now the actual virtual function call must be performed. First, the VPTR must be produced, so the VTABLE can be found. For this compiler the VPTR is inserted at the beginning of the object, so the contents of **this** correspond to the VPTR. The line

```
    mov bx, word ptr [si]
```

fetches the word that **si** (that is, **this**) points to, which is the VPTR. It places the VPTR into the register **bx**.

The VPTR contained in **bx** points to the starting address of the VTABLE, but the function pointer to call isn't at location zero of the VTABLE, but instead at location two (because it's the third function in the list). For this memory model each function pointer is two bytes long, so the compiler adds four to the VPTR to calculate where the address of the proper function is. Note that this is a constant value, established at compile time, so the only thing that matters is that the function pointer at location number two is the one for **adjust( )**. Fortunately, the compiler takes care of all the bookkeeping for you and ensures that all the function pointers in all the VTABLEs of a particular class hierarchy occur in the same order, regardless of the order that you may override them in derived classes.

Once the address of the proper function pointer in the VTABLE is calculated, that function is called. So the address is fetched and called all at once in the statement

```
    call word ptr [bx+4]
```

Finally, the stack pointer is moved back up to clean off the arguments that were pushed before the call. In C and C++ assembly code you'll often see the caller clean off the arguments but this may vary depending on processors and compiler implementations.

Installing the vpointer

Because the VPTR determines the virtual function behavior of the object, you can see how it's critical that the VPTR always be pointing to the proper VTABLE. You don't ever want to be able to make a call to a virtual function before the VPTR is properly initialized. Of course, the place where initialization can be guaranteed is in the constructor, but none of the **Instrument** examples has a constructor.

This is where creation of the default constructor is essential. In the **Instrument** examples, the compiler creates a default constructor that does nothing except initialize the VPTR. This constructor, of course, is automatically called for all **Instrument** objects before you can do anything with them, so you know that it's always safe to call virtual functions.

The implications of the automatic initialization of the VPTR inside the constructor are discussed in a later section.

Objects are different

It's important to realize that upcasting deals only with addresses. If the compiler has an object, it knows the exact type and therefore (in C++) will not use late binding for any function calls – or at least, the compiler doesn't *need* to use late binding. For efficiency's sake, most compilers will perform early binding when they are making a call to a virtual function for an object because they know the exact type. Here's an example:

```
//: C15:Early.cpp
// Early binding & virtual functions
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
  string speak() const { return "Bark!"; }
};
```

```
int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding for both:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probably):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```

In **p1–>speak( )** and **p2.speak( )**, addresses are used, which means the information is incomplete: **p1** and **p2** can represent the address of a **Pet** *or* something derived from **Pet**, so the virtual mechanism must be used. When calling **p3.speak( )** there's no ambiguity. The compiler knows the exact type and that it's an object, so it can't possibly be an object derived from **Pet** – it's *exactly* a **Pet**. Thus, early binding is probably used. However, if the compiler doesn't want to work so hard, it can still use late binding and the same behavior will occur.

Why virtual functions?

At this point you may have a question: "If this technique is so important, and if it makes the 'right' function call all the time, why is it an option? Why do I even need to know about it?"

This is a good question, and the answer is part of the fundamental philosophy of C++: "Because it's not quite as efficient." You can see from the previous assembly-language output that instead of one simple CALL to an absolute address, there are two – more sophisticated – assembly instructions required to set up the virtual function call. This requires both code space and execution time.

Some object-oriented languages have taken the approach that late binding is so intrinsic to object-oriented programming that it should always take place, that it should not be an option, and the user shouldn't have to know about it. This is a design decision when creating a language, and that particular path is appropriate for many languages.[56] However, C++ comes from the C heritage, where efficiency is critical. After all, C was created to replace assembly language for the implementation of an operating system (thereby rendering that operating system – Unix – far more portable than its predecessors). One of the main reasons for the invention of C++ was to make C programmers more efficient.[57] And the first question asked when C programmers encounter C++ is, "What kind of size and speed impact will I get?" If the answer were, "Everything's great except for function calls when you'll always have a little extra overhead," many people would stick with C rather than make the change to C++. In addition, inline functions would not be possible, because virtual functions must have an address to put into the VTABLE. So the virtual function is an option, *and* the language defaults to nonvirtual, which is the fastest configuration. Stroustrup stated that his guideline was, "If you don't use it, you don't pay for it."

Thus, the **virtual** keyword is provided for efficiency tuning. When designing your classes, however, you shouldn't be worrying about efficiency tuning. If you're going to use polymorphism, use virtual functions everywhere. You only need to look for functions that can be made non-virtual when searching for ways to speed up your code (and there are usually much bigger gains to be had in other areas – a good profiler will do a better job of finding bottlenecks than you will by making guesses).

Anecdotal evidence suggests that the size and speed impacts of going to C++ are within 10 percent of the size and speed of C, and often much closer to the same. The reason you might get better size and speed efficiency is because you may design a C++ program in a smaller, faster way than you would using C.

Abstract base classes and pure virtual functions

Often in a design, you want the base class to present *only* an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class *abstract*, which happens if you give it at least one *pure virtual function*. You can recognize a pure virtual function because it uses the **virtual** keyword and is followed by **= 0**. If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool that allows you to enforce a particular design.

When an abstract class is inherited, all pure virtual functions must be implemented, or the inherited class becomes abstract as well. Creating a pure virtual function allows you to put a member function in an interface without being forced to provide a possibly meaningless body of code for that member function. At the same time, a pure virtual function forces inherited classes to provide a definition for it.

In all of the instrument examples, the functions in the base class **Instrument** were always "dummy" functions. If these functions are ever called, something is wrong. That's because the intent of **Instrument** is to create a common interface for all of the classes derived from it.

The only reason to establish the common interface is so it can be expressed differently for each different subtype. It creates a basic form that determines what's in common with all of the derived classes – nothing else. So **Instrument** is an appropriate candidate to be an abstract class. You create an abstract class when you only want to manipulate a set of classes through a common interface, but the common interface doesn't need to have an implementation (or at least, a full implementation).

If you have a concept like **Instrument** that works as an abstract class, objects of that class almost always have no meaning. That is, **Instrument** is meant to express only the interface, and not a particular implementation, so creating an object that is only an **Instrument** makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the virtual functions in **Instrument** print error messages, but that delays the appearance of the error information until runtime and it requires reliable exhaustive testing on the part of the user. It is much better to catch the problem at compile time.

Here is the syntax used for a pure virtual declaration:

```
    virtual void f() = 0;
```

By doing this, you tell the compiler to reserve a slot for a function in the VTABLE, but not to put an address in that particular slot. Even if only one function in a class is declared as pure virtual, the VTABLE is incomplete.

If the VTABLE for a class is incomplete, what is the compiler supposed to do when someone tries to make an object of that class? It cannot safely create an object of an abstract class, so you get an error message from the compiler. Thus, the compiler guarantees the purity of the abstract class. By making a class abstract, you ensure that the client programmer cannot misuse it.

Here's **Instrument4.cpp** modified to use pure virtual functions. Because the class has nothing but pure virtual functions, we call it a *pure abstract class*:

```
//: C15:Instrument5.cpp
// Pure abstract base classes
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};
```

```
class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Pure virtual functions are helpful because they make explicit the abstractness of a class and tell both the user and the compiler how it was intended to be used.

Note that pure virtual functions prevent an abstract class from being passed into a function *by value*. Thus, it is also a way to prevent *object slicing* (which will be described shortly). By making a class abstract, you can ensure that a pointer or reference is always used during upcasting to that class.

Just because one pure virtual function prevents the VTABLE from being completed doesn’t mean that you don’t want function bodies for some of the others. Often you will want to call a base-class version of a function, even if it is virtual. It’s always a good idea to put common code as close as possible to the root of your hierarchy. Not only does this save code space, it allows easy propagation of changes.

Pure virtual definitions

It’s possible to provide a definition for a pure virtual function in the base class. You’re still telling the compiler not to allow objects of that abstract base class, and the pure virtual functions must still be defined in derived classes in order to create objects. However, there may be a common piece of code that you want some or all of the derived class definitions to call rather than duplicating that code in every function.

Here’s what a pure virtual definition looks like:

```
//: C15:PureVirtualDefinitions.cpp
// Pure virtual base definitions
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;
  // Inline pure virtual definitions illegal:
  //!  virtual void sleep() const = 0 {}
};

// OK, not defined inline
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
  // Use the common Pet code:
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};

int main() {
  Dog simba;  // Richard's dog
  simba.speak();
  simba.eat();
} ///:~
```

The slot in the **Pet** VTABLE is still empty, but there happens to be a function by that name that you can call in the derived class.

The other benefit to this feature is that it allows you to change from an ordinary virtual to a pure virtual without disturbing the existing code. (This is a way for you to locate classes that don't override that virtual function.)

Inheritance and the VTABLE

You can imagine what happens when you perform inheritance and override some of the virtual functions. The compiler creates a new VTABLE for your new class, and it inserts your new function addresses using the base-class function addresses for any virtual functions you don't override. One way or another, for every object that can be created (that is, its class has no pure virtuals) there's always a full set of function addresses in the VTABLE, so you'll never be able to make a call to an address that isn't there (which would be disastrous).

But what happens when you inherit and add new virtual functions in the *derived* class? Here's a simple example:

```
//: C15:AddingVirtuals.cpp
// Adding virtuals in derivation
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& petName) : pname(petName) {}
  virtual string name() const { return pname; }
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
  string name;
public:
  Dog(const string& petName) : Pet(petName) {}
  // New virtual function in the Dog class:
  virtual string sit() const {
    return Pet::name() + " sits";
  }
  string speak() const { // Override
    return Pet::name() + " says 'Bark!'";
  }
};

int main() {
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
  cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
//! cout << "p[1]->sit() = "
//!      << p[1]->sit() << endl; // Illegal
} ///:~
```

The class **Pet** contains a two virtual functions: **speak( )** and **name( )**. **Dog** adds a third virtual function called **sit( )**, as well as overriding the meaning of **speak( )**. A diagram will help you visualize what's happening. Here are the VTABLEs created by the compiler for **Pet** and **Dog**:

Notice that the compiler maps the location of the **speak( )** address into exactly the same spot in the **Dog** VTABLE as it is in the **Pet** VTABLE. Similarly, if a class **Pug** is inherited from **Dog**, its version of **sit( )** would be placed in its VTABLE in exactly the same spot as it is in **Dog**. This is because (as you saw with the assembly-language example) the compiler generates code that uses a simple numerical offset into the VTABLE to select the virtual function. Regardless of the specific subtype the object belongs to, its VTABLE is laid out the same way, so calls to the virtual functions will always be made the same way.

In this case, however, the compiler is working only with a pointer to a base-class object. The base class has only the **speak( )** and **name( )** functions, so those is the only functions the compiler will allow you to call. How could it possibly know that you are working with a **Dog** object, if it has only a pointer to a base-class object? That pointer might point to some other type, which doesn't have a **sit( )** function. It may or may not have some other function address at that point in the VTABLE, but in either case, making a virtual call to that VTABLE address is not what you want to do. So the compiler is doing its job by protecting you from making virtual calls to functions that exist only in derived classes.

There are some less-common cases in which you may know that the pointer actually points to an object of a specific subclass. If you want to call a function that only exists in that subclass, then you must cast the pointer. You can remove the error message produced by the previous program like this:

```
        ((Dog*)p[1])->sit()
```

Here, you happen to know that **p[1]** points to a **Dog** object, but in general you don't know that. If your problem is set up so that you must know the exact types of all objects, you should rethink it, because you're probably not using virtual functions properly. However, there are some situations in which the design works best (or you have no choice) if you know the exact type of all objects kept in a generic container. This is the problem of *run-time type identification* (RTTI)**.**

RTTI is all about casting base-class pointers *down* to derived-class pointers ("up" and "down" are relative to a typical class diagram, with the base class at the top). Casting *up* happens automatically, with no coercion, because it's completely safe. Casting *down* is unsafe because there's no compile time information about the actual types, so you must know exactly what type the object is. If you cast it into the wrong type, you'll be in trouble.

RTTI is described later in this chapter, and Volume 2 of this book has a chapter devoted to the subject.

Object slicing

There is a distinct difference between passing the addresses of objects and passing objects by value when using polymorphism. All the examples you've seen here, and virtually all the examples you should see, pass addresses and not values. This is because addresses all have the same size[58], so passing the address of an object of a derived type (which is usually a bigger object) is the same as passing the address of an object of the base type (which is usually a smaller object). As explained before, this is the goal when using polymorphism – code that manipulates a base type can transparently manipulate derived-type objects as well.

If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is "sliced" until all that remains is the subobject that corresponds to the destination type of your cast. In the following example you can see what happens when an object is sliced:

```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // Slices the object
  cout << p.description() << endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```

The function **describe( )** is passed an object of type **Pet** *by value*. It then calls the virtual function **description( )** for the **Pet** object. In **main( )**, you might expect the first call to produce "This is Alfred," and the second to produce "Fluffy likes to sleep." In fact, both calls use the base-class version of **description( )**.

Two things are happening in this program. First, because **describe( )** accepts a **Pet** *object* (rather than a pointer or reference), any calls to **describe( )** will cause an object the size of **Pet** to be pushed on the stack and cleaned up after the call. This means that if an object of a class inherited from **Pet** is passed to **describe( )**, the compiler accepts it, but it copies only the **Pet** portion of the object. It *slices* the derived portion off of the object, like this:

Now you may wonder about the virtual function call. **Dog::description( )** makes use of portions of both **Pet** (which still exists) and **Dog**, which no longer exists because it was sliced off! So what happens when the virtual function is called?

You're saved from disaster because the object is being passed by value. Because of this, the compiler knows the precise type of the object because the derived object has been forced to become a base object. When passing by value, the copy-constructor for a **Pet** object is used, which initializes the VPTR to the **Pet** VTABLE and copies only the **Pet** parts of the object. There's no explicit copy-constructor here, so the compiler synthesizes one. Under all interpretations, the object truly becomes a **Pet** during slicing.

Object slicing actually removes part of the existing object as it copies it into the new object, rather than simply changing the meaning of an address as when using a pointer or reference. Because of this, upcasting into an object is not done often; in fact, it's usually something to watch out for and prevent. Note that, in this example, if **description( )** were made into a pure virtual function in the base class (which is not unreasonable, since it doesn't really do anything in the base class), then the compiler would prevent object slicing because that wouldn't allow you to "create" an object of the base type (which is what happens when you upcast by value). This could be the most important value of pure virtual functions: to prevent object slicing by generating a compile-time error message if someone tries to do it.

Overloading & overriding

In Chapter 14, you saw that redefining an overloaded function in the base class hides all of the other base-class versions of that function. When **virtual** functions are involved the behavior is a little different. Consider a modified version of the **NameHiding.cpp** example from Chapter 14:

```
//: C15:NameHiding2.cpp
// Virtual functions restrict overloading
#include <iostream>
#include <string>
```

```
using namespace std;

class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Overriding a virtual function:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // Cannot change return type:
  //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
  // Change argument list:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // string version hidden
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // f() version hidden
//!  d4.f(s); // string version hidden
  Base& br = d4; // Upcast
//!  br.f(1); // Derived version unavailable
  br.f(); // Base version available
  br.f(s); // Base version abailable
} ///:~
```

The first thing to notice is that in **Derived3**, the compiler will not allow you to change the return type of an overridden function (it will allow it if **f( )** is not virtual). This is an important restriction because the compiler must guarantee that you can polymorphically call the function through the base class, and if the base class is expecting an **int** to be returned from **f( )**, then the derived-class version of **f( )** must keep that contract or else things will break.

The rule shown in Chapter 14 still works: if you override one of the overloaded member functions in the base class, the other overloaded versions become hidden in the derived class. In **main( )** the code that tests **Derived4** shows that this happens even if the new version of **f( )** isn't actually overriding an existing virtual function interface – both of the base-class versions of **f( )** are hidden by **f(int)**. However, if you upcast **d4** to **Base**, then only the base-class versions are available (because that's what the base-class contract promises) and the derived-class version is not available (because it isn't specified in the base class).

Variant return type

The **Derived3** class above suggests that you cannot modify the return type of a virtual function during overriding. This is generally true, but there is a special case in which you can slightly modify the return type. If you're returning a pointer or a reference to a base class, then the overridden version of the function may return a pointer or reference to a class derived from what the base returns. For example:

```
//: C15:VariantReturn.cpp
// Returning a pointer or reference to a derived
// type during ovverriding
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
  virtual string foodType() const = 0;
};
```

```
class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast to base type:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};

class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Return exact type instead:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};

int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Can return the exact type:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Cannot return the exact type:
//!  bf = b.eats();
  // Must downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

The **Pet::eats( )** member function returns a pointer to a **PetFood**. In **Bird**, this member function is overloaded exactly as in the base class, including the return type. That is, **Bird::eats( )** upcasts the **BirdFood** to a **PetFood**.

But in **Cat**, the return type of **eats( )** is a pointer to **CatFood**, a type derived from **PetFood**. The fact that the return type is inherited from the return type of the base-class function is the only reason this compiles. That way, the contract is still fulfilled; **eats( )** always returns a **PetFood** pointer.

If you think polymorphically, this doesn't seem necessary. Why not just upcast all the return types to **PetFood***, just as **Bird::eats( )** did? This is typically a good solution, but at the end of **main( )**, you see the difference: **Cat::eats( )** can return the exact type of **PetFood**, whereas the return value of **Bird::eats( )** must be downcast to the exact type.

So being able to return the exact type is a little more general, and doesn't lose the specific type information by automatically upcasting. However, returning the base type will generally solve your problems so this is a rather specialized feature.

virtual functions & constructors

When an object containing virtual functions is created, its VPTR must be initialized to point to the proper VTABLE. This must be done before there's any possibility of calling a virtual function. As you might guess, because the constructor has the job of bringing an object into existence, it is also the constructor's job to set up the VPTR. The compiler secretly inserts code into the beginning of the constructor that initializes the VPTR. And as described in Chapter 14, if you don't explicitly create a constructor for a class, the compiler will synthesize one for you. If the class has virtual functions, the synthesized constructor will include the proper VPTR initialization code. This has several implications.

The first concerns efficiency. The reason for **inline** functions is to reduce the calling overhead for small functions. If C++ didn't provide **inline** functions, the preprocessor might be used to create these "macros." However, the preprocessor has no concept of access or classes, and therefore couldn't be used to create member function macros. In addition, with constructors that must have hidden code inserted by the compiler, a preprocessor macro wouldn't work at all.

You must be aware when hunting for efficiency holes that the compiler is inserting hidden code into your constructor function. Not only must it initialize the VPTR, it must also check the value of **this** (in case the **operator new** returns zero) and call base-class constructors. Taken together, this code can impact what you thought was a tiny inline function call. In particular, the size of the constructor may overwhelm the savings you get from reduced function-call overhead. If you make a lot of inline constructor calls, your code size can grow without any benefits in speed.

Of course, you probably won’t make all tiny constructors non-inline right away, because they’re much easier to write as inlines. But when you’re tuning your code, remember to consider removing the inline constructors.

Order of constructor calls

The second interesting facet of constructors and virtual functions concerns the order of constructor calls and the way virtual calls are made within constructors.

All base-class constructors are always called in the constructor for an inherited class. This makes sense because the constructor has a special job: to see that the object is built properly. A derived class has access only to its own members, and not those of the base class. Only the base-class constructor can properly initialize its own elements. Therefore it’s essential that all constructors get called; otherwise the entire object wouldn’t be constructed properly. That’s why the compiler enforces a constructor call for every portion of a derived class. It will call the default constructor if you don’t explicitly call a base-class constructor in the constructor initializer list. If there is no default constructor, the compiler will complain.

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any **public** and **protected** members of the base class. This means you must be able to assume that all the members of the base class are valid when you’re in the derived class. In a normal member function, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all members that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then when you’re in the derived-class constructor, all the members you can access in the base class have been initialized. “Knowing all members are valid” inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) in the constructor initializer list. If you follow this practice, you can assume that all base class members *and* member objects of the current object have been initialized.

Behavior of virtual functions inside constructors

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a virtual function? Inside an ordinary member function you can imagine what will happen – the virtual call is resolved at runtime because the object cannot know whether it belongs to the class the member function is in, or some class derived from it. For consistency, you might think this is what should happen inside constructors.

This is not the case. If you call a virtual function inside a constructor, only the local version of the function is used. That is, the virtual mechanism doesn't work within the constructor.

This behavior makes sense for two reasons. Conceptually, the constructor's job is to bring the object into existence (which is hardly an ordinary feat). Inside any constructor, the object may only be partially formed – you can only know that the base-class objects have been initialized, but you cannot know which classes are inherited from you. A virtual function call, however, reaches "forward" or "outward" into the inheritance hierarchy. It calls a function in a derived class. If you could do this inside a constructor, you'd be calling a function that might manipulate members that hadn't been initialized yet, a sure recipe for disaster.

The second reason is a mechanical one. When a constructor is called, one of the first things it does is initialize its VPTR. However, it can only know that it is of the "current" type – the type the constructor was written for. The constructor code is completely ignorant of whether or not the object is in the base of another class. When the compiler generates code for that constructor, it generates code for a constructor of that class, not a base class and not a class derived from it (because a class can't know who inherits it). So the VPTR it uses must be for the VTABLE of *that* class. The VPTR remains initialized to that VTABLE for the rest of the object's lifetime *unless* this isn't the last constructor call. If a more-derived constructor is called afterwards, that constructor sets the VPTR to *its* VTABLE, and so on, until the last constructor finishes. The state of the VPTR is determined by the constructor that is called last. This is another reason why the constructors are called in order from base to most-derived.

But while all this series of constructor calls is taking place, each constructor has set the VPTR to its own VTABLE. If it uses the virtual mechanism for function calls, it will produce only a call through its own VTABLE, not the most-derived VTABLE (as would be the case after *all* the constructors were called). In addition, many compilers recognize that a virtual function call is being made inside a constructor, and perform early binding because they know that late-binding will produce a call only to the local function. In either event, you won't get the results you might initially expect from a virtual function call inside a constructor.

Destructors and virtual destructors

You cannot use the **virtual** keyword with constructors, but destructors can and often must be virtual.

The constructor has the special job of putting an object together piece-by-piece, first by calling the base constructor, then the more derived constructors in order of inheritance (it must also call member-object constructors along the way). Similarly, the destructor has a special job: it must disassemble an object that may belong to a hierarchy of classes. To do this, the compiler generates code that calls all the destructors, but in the *reverse* order that they are called by the constructor. That is, the destructor starts at the most-derived class and works its way down to the base class. This is the safe and desirable thing to do because the current destructor can always know that the base-class members are alive and active. If you need to call a base-class member function inside your destructor, it is safe to do so. Thus, the destructor can perform its own cleanup, then call the next-down destructor, which will perform *its* own cleanup, etc. Each destructor knows what its class is derived *from*, but not what is derived from it.

You should keep in mind that constructors and destructors are the only places where this hierarchy of calls must happen (and thus the proper hierarchy is automatically generated by the compiler). In all other functions, only *that* function will be called (and not base-class versions), whether it's virtual or not. The only way for base-class versions of the same function to be called in ordinary functions (virtual or not) is if you *explicitly* call that function.

Normally, the action of the destructor is quite adequate. But what happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? This activity is a major objective in object-oriented programming. The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new**. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during **delete**. Sound familiar? This is the same problem that virtual functions were created to solve for the general case. Fortunately, virtual functions work for destructors as they do for all other functions except constructors.

```
//: C15:VirtualDestructors.cpp
// Behavior of virtual vs. non-virtual destructor
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
} ///:~
```

When you run the program, you'll see that **delete bp** only calls the base-class destructor, while **delete b2p** calls the derived-class destructor followed by the base-class destructor, which is the behavior we desire. Forgetting to make a destructor **virtual** is an insidious bug because it often doesn't directly affect the behavior of your program, but it can quietly introduce a memory leak. Also, the fact that *some* destruction is occurring can further mask the problem.

Even though the destructor, like the constructor, is an "exceptional" function, it is possible for the destructor to be virtual because the object already knows what type it is (whereas it doesn't during construction). Once an object has been constructed, its VPTR is initialized, so virtual function calls can take place.

Pure virtual destructors

While pure virtual destructors are legal in Standard C++, there is an added constraint when using them: you must provide a function body for the pure virtual destructor. This seems counterintuitive; how can a virtual function be "pure" if it needs a function body? But if you keep in mind that constructors and destructors are special operations it makes more sense, especially if you remember that all destructors in a class hierarchy are always called. If you *could* leave off the definition for a pure virtual destructor, what function body would be called during destruction? Thus, it's absolutely necessary that the compiler and linker enforce the existence of a function body for a pure virtual destructor.

If it's pure, but it has to have a function body, what's the value of it? The only difference you'll see between the pure and non-pure virtual destructor is that the pure virtual destructor does cause the base class to be abstract, so you cannot create an object of the base class (although this would also be true if any other member function of the base class were pure virtual).

Things are a bit confusing, however, when you inherit a class from one that contains a pure virtual destructor. Unlike every other pure virtual function, you are *not* required to provide a definition of a pure virtual destructor in the derived class. The fact that the following compiles and links is the proof:

```
//: C15:UnAbstract.cpp
// Pure virtual destructors
// seem to behave strangely

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?

int main() { Derived d; } ///:~
```

Normally, a pure virtual function in a base class would cause the derived class to be abstract unless it (and all other pure virtual functions) is given a definition. But here, this seems not to be the case. However, remember that the compiler *automatically* creates a destructor definition for every class if you don't create one. That's what's happening here – the base class destructor is being quietly overridden, and thus the definition is being provided by the compiler and **Derived** is not actually abstract.

This brings up an interesting question: What is the point of a pure virtual destructor? Unlike an ordinary pure virtual function, you *must* give it a function body. In a derived class, you aren't forced to provide a definition since the compiler synthesizes the destructor for you. So what's the difference between a regular virtual destructor and a pure virtual destructor?

The only distinction occurs when you have a class that only has a single pure virtual function: the destructor. In this case, the only effect of the purity of the destructor is to prevent the instantiation of the base class. If there were any other pure virtual functions, they would prevent the instantiation of the base class, but if there are no others, then the pure virtual destructor will do it. So, while the addition of a virtual destructor is essential, whether it's pure or not isn't so important.

When you run the following example, you can see that the pure virtual function body is called after the derived class version, just as with any other destructor:

```
//: C15:PureVirtualDestructors.cpp
// Pure virtual destructors
// require a function body
#include <iostream>
using namespace std;

class Pet {
public:
  virtual ~Pet() = 0;
};

Pet::~Pet() {
  cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};

int main() {
  Pet* p = new Dog; // Upcast
  delete p; // Virtual destructor call
} ///:~
```

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Virtuals in destructors

There's something that happens during destruction that you might not immediately expect. If you're inside an ordinary member function and you call a virtual function, that function is called using the late-binding mechanism. This is not true with destructors, virtual or not. Inside a destructor, only the "local" version of the member function is called; the virtual mechanism is ignored.

```
//: C15:VirtualsInDestructors.cpp
// Virtual calls inside destructors
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "Base1()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Upcast
  delete bp;
} ///:~
```

During the destructor call, **Derived::f( )** is *not* called, even though **f( )** is virtual.

Why is this? Suppose the virtual mechanism *were* used inside the destructor. Then it would be possible for the virtual call to resolve to a function that was "farther out" (more derived) on the inheritance hierarchy than the current destructor. But destructors are called from the "outside in" (from the most-derived destructor down to the base destructor), so the actual function called would rely on portions of an object that have *already been destroyed*! Instead, the compiler resolves the calls at compile-time and calls only the "local" version of the function. Notice that the same is true for the constructor (as described earlier), but in the constructor's case the type information wasn't available, whereas in the destructor the information (that is, the VPTR) is there, but is isn't reliable.

Creating an object-based hierarchy

An issue that has been recurring throughout this book during the demonstration of the container classes **Stack** and **Stash** is the "ownership problem." The "owner" refers to who or what is responsible for calling **delete** for objects that have been created dynamically (using **new**). The problem when using containers is that they need to be flexible enough to hold different types of objects. To do this, the containers have held **void** pointers and so they haven't known the type of object they've held. Deleting a **void** pointer doesn't call the destructor, so the container couldn't be responsible for cleaning up its objects.

One solution was presented in the example **C14:InheritStack.cpp**, in which the **Stack** was inherited into a new class that accepted and produced only **string** pointers. Since it knew that it could hold only pointers to **string** objects, it could properly delete them. This was a nice solution, but it requires you to inherit a new container class for each type that you want to hold in the container. (Although this seems tedious now, it will actually work quite well in Chapter 16, when templates are introduced.)

The problem is that you want the container to hold more than one type, but you don't want to use **void** pointers. Another solution is to use polymorphism by forcing all the objects held in the container to be inherited from the same base class. That is, the container holds the objects of the base class, and then you can call virtual functions – in particular, you can call virtual destructors to solve the ownership problem.

This solution uses what is referred to as a *singly-rooted hierarchy* or an *object-based hierarchy* (because the root class of the hierarchy is usually named "Object"). It turns out that there are many other benefits to using a singly-rooted hierarchy; in fact, every other object-oriented language but C++ enforces the use of such a hierarchy – when you create a class, you are automatically inheriting it directly or indirectly from a common base class, a base class that was established by the creators of the language. In C++, it was thought that the enforced use of this common base class would cause too much overhead, so it was left out. However, you can choose to use a common base class in your own projects, and this subject will be examined further in Volume 2 of this book.

To solve the ownership problem, we can create an extremely simple **Object** for the base class, which contains only a virtual destructor. The **Stack** can then hold classes inherited from **Object**:

```
//: C15:OStack.h
// Using a singly-rooted hierarchy
#ifndef OSTACK_H
#define OSTACK_H
```

```
class Object {
public:
  virtual ~Object() = 0;
};

// Required definition:
inline Object::~Object() {}

class Stack {
  struct Link {
    Object* data;
    Link* next;
    Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(Object* dat) {
    head = new Link(dat, head);
  }
  Object* peek() const {
    return head ? head->data : 0;
  }
  Object* pop() {
    if(head == 0) return 0;
    Object* result = head->data;
```

```
        Link* oldHead = head;
        head = head->next;
        delete oldHead;
        return result;
      }
    };
    #endif // OSTACK_H ///:~
```

To simplify things by keeping everything in the header file, the (required) definition for the pure virtual destructor is inlined into the header file, and **pop( )** (which might be considered too large for inlining) is also inlined.

Link objects now hold pointers to **Object** rather than **void** pointers, and the **Stack** will only accept and return **Object** pointers. Now **Stack** is much more flexible, since it will hold lots of different types but will also destroy any objects that are left on the **Stack**. The new limitation (which will be finally removed when templates are applied to the problem in Chapter 16) is that anything that is placed on the **Stack** must be inherited from **Object**. That's fine if you are starting your class from scratch, but what if you already have a class such as **string** that you want to be able to put onto the **Stack**? In this case, the new class must be both a **string** and an **Object**, which means it must be inherited from both classes. This is called *multiple inheritance* and it is the subject of an entire chapter in Volume 2 of this book (downloadable from *www.BruceEckel.com*). When you read that chapter, you'll see that multiple inheritance can be fraught with complexity, and is a feature you should use sparingly. In this situation, however, everything is simple enough that we don't trip across any multiple inheritance pitfalls:

```
    //: C15:OStackTest.cpp
    //{T} OStackTest.cpp
    #include "OStack.h"
    #include "../require.h"
    #include <fstream>
    #include <iostream>
```

```
#include <string>
using namespace std;

// Use multiple inheritance. We want
// both a string and an Object:
class MyString: public string, public Object {
public:
  ~MyString() {
    cout << "deleting string: " << *this << endl;
  }
  MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new MyString(line));
  // Pop some lines from the stack:
  MyString* s;
  for(int i = 0; i < 10; i++) {
    if((s=(MyString*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  }
  cout << "Letting the destructor do the rest:"
```

```
        << endl;
} ///:~
```

Although this is similar to the previous version of the test program for **Stack**, you'll notice that only 10 elements are popped from the stack, which means there are probably some objects remaining. Because the **Stack** knows that it holds **Object**s, the destructor can properly clean things up, and you'll see this in the output of the program, since the **MyString** objects print messages as they are destroyed.

Creating containers that hold **Object**s is not an unreasonable approach – *if* you have a singly-rooted hierarchy (enforced either by the language or by the requirement that every class inherit from **Object**). In that case, everything is guaranteed to be an **Object** and so it's not very complicated to use the containers. In C++, however, you cannot expect this from every class, so you're bound to trip over multiple inheritance if you take this approach. You'll see in Chapter 16 that templates solve the problem in a much simpler and more elegant fashion.

Operator overloading

You can make operators **virtual** just like other member functions. Implementing **virtual** operators often becomes confusing, however, because you may be operating on two objects, both with unknown types. This is usually the case with mathematical components (for which you often overload operators). For example, consider a system that deals with matrices, vectors and scalar values, all three of which are derived from class **Math**:

```
//: C15:OperatorPolymorphism.cpp
// Polymorphism with overloaded operators
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual Math& multiply(Scalar*) = 0;
  virtual Math& multiply(Vector*) = 0;
  virtual ~Math() {}
};

class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};
```

```
class Scalar : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};

class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      m1 * m2;
    }
} ///:~
```

For simplicity, only the **operator*** has been overloaded. The goal is to be able to multiply any two **Math** objects and produce the desired result – and note that multiplying a matrix by a vector is a very different operation than multiplying a vector by a matrix.

The problem is that, in **main( )**, the expression **m1** * **m2** contains two upcast **Math** references, and thus two objects of unknown type. A virtual function is only capable of making a single dispatch – that is, determining the type of one unknown object. To determine both types a technique called *multiple dispatching* is used in this example, whereby what appears to be a single virtual function call results in a second virtual call. By the time this second call is made, you've determined both types of object, and can perform the proper activity. It's not transparent at first, but if you stare at the example for awhile it should begin to make sense. This topic is explored in more depth in the Design Patterns chapter in Volume 2, which you can download at *www.BruceEckel.com*.

Downcasting

As you might guess, since there's such a thing as upcasting – moving up an inheritance hierarchy – there should also be *downcasting* to move down a hierarchy. But upcasting is easy since as you move up an inheritance hierarchy the classes always converge to more general classes. That is, when you upcast you are always clearly derived from an ancestor class (typically only one, except in the case of multiple inheritance) but when you downcast there are usually several possibilities that you could cast to. More specifically, a **Circle** is a type of **Shape** (that's the upcast), but if you try to downcast a **Shape** it could be a **Circle**, **Square**, **Triangle**, etc. So the dilemma is figuring out a way to safely downcast. (But an even more important issue is asking yourself why you're downcasting in the first place instead of just using polymorphism to automatically figure out the correct type. The avoidance of downcasting is covered in Volume 2 of this book.)

C++ provides a special *explicit cast* (introduced in Chapter 3) called **dynamic_cast** that is a *type-safe downcast* operation. When you use **dynamic_cast** to try to cast down to a particular type, the return value will be a pointer to the desired type only if the cast is proper and successful, otherwise it will return zero to indicate that this was not the correct type. Here's a minimal example:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Try to cast it to Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Try to cast it to Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

When you use **dynamic_cast**, you must be working with a true polymorphic hierarchy – one with virtual functions – because **dynamic_cast** uses information stored in the VTABLE to determine the actual type. Here, the base class contains a virtual destructor and that suffices. In **main( )**, a **Cat** pointer is upcast to a **Pet**, and then a downcast is attempted to both a **Dog** pointer and a **Cat** pointer. Both pointers are printed, and you'll see when you run the program that the incorrect downcast produces a zero result. Of course, whenever you downcast you are responsible for checking to make sure that the result of the cast is nonzero. Also, you should not assume that the pointer will be exactly the same, because sometimes pointer adjustments take place during upcasting and downcasting (in particular, with multiple inheritance).

A **dynamic_cast** requires a little bit of extra overhead to run; not much, but if you're doing a lot of **dynamic_cast**ing (in which case you should be seriously questioning your program design) this may become a performance issue. In some cases you may know something special during downcasting that allows you to say for sure what type you're dealing with, in which case the extra overhead of the **dynamic_cast** becomes unnecessary, and you can use a **static_cast** instead. Here's how it might work:

```
  //: C15:StaticHierarchyNavigation.cpp
  // Navigating class hierarchies with static_cast
  #include <iostream>
  #include <typeinfo>
  using namespace std;
```

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);
  // (Since upcasting is such a safe and common
  // operation, the cast becomes cluttering)
  Circle* cp = 0;
  Square* sp = 0;
  // Static Navigation of class hierarchies
  // requires extra type information:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an efficiency hack;
  // dynamic_cast is always safer. However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error message, while
  Other* op2 = (Other*)s;
  // does not
} ///:~
```

In this program, a new feature is used that is not fully described until Volume 2 of this book, where a chapter is given to the topic: C++'s *run-time type information* (RTTI) mechanism. RTTI allows you to discover type information that has been lost by upcasting. The **dynamic_cast** is actually one form of RTTI. Here, the **typeid** keyword (declared in the header file **<typeinfo>**) is used to detect the types of the pointers. You can see that the type of the upcast **Shape** pointer is successively compared to a **Circle** pointer and a **Square** pointer to see if there's a match. There's more to RTTI than **typeid**, and you can also imagine that it would be fairly easy to implement your own type information system using a virtual function.

A **Circle** object is created and the address is upcast to a **Shape** pointer; the second version of the expression shows how you can use **static_cast** to be more explicit about the upcast. However, since an upcast is always safe and it's a common thing to do, I consider an explicit cast for upcasting to be cluttering and unnecessary.

RTTI is used to determine the type, and then **static_cast** is used to perform the downcast. But notice that in this design the process is effectively the same as using **dynamic_cast**, and the client programmer must do some testing to discover the cast that was actually successful. You'll typically want a situation that's more deterministic than in the example above before using **static_cast** rather than **dynamic_cast** (and, again, you want to carefully examine your design before using **dynamic_cast**).

If a class hierarchy has no **virtual** functions (which is a questionable design) or if you have other information that allows you to safely downcast, it's a tiny bit faster to do the downcast statically than with **dynamic_cast**. In addition, **static_cast** won't allow you to cast out of the hierarchy, as the traditional cast will, so it's safer. However, statically navigating class hierarchies is always risky and you should use **dynamic_cast** unless you have a special situation.

Summary

Polymorphism – implemented in C++ with virtual functions – means "different forms." In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the virtual functions.

You've seen in this chapter that it's impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like **const** or a **switch** statement, for example), but instead works only in concert, as part of a "big picture" of class relationships. People are often confused by other, non-object-oriented features of C++, like overloading and default arguments, which are sometimes presented as object-oriented. Don't be fooled; if it isn't late binding, it isn't polymorphism.

To use polymorphism – and thus, object-oriented techniques – effectively in your programs you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle, because the results are faster program development, better code organization, extensible programs, and easier code maintenance.

Polymorphism completes the object-oriented features of the language, but there are two more major features in C++: templates (which are introduced in Chapter 16 and covered in much more detail in Volume 2), and exception handling (which is covered in Volume 2). These features provide you as much increase in programming power as each of the object-oriented features: abstract data typing, inheritance, and polymorphism.

Exercises

Solutions to selected exercises can be found in the electronic document *The Thinking in C++ Annotated Solution Guide*, available for a small fee from www.BruceEckel.com.

1. Create a simple "shape" hierarchy: a base class called **Shape** and derived classes called **Circle**, **Square**, and **Triangle**. In the base class, make a virtual function called **draw( ),** and override this in the derived classes. Make an array of pointers to **Shape** objects that you create on the heap (and thus perform upcasting of the pointers), and call **draw( )** through the base-class pointers, to verify the behavior of the

virtual function. If your debugger supports it, single-step through the code.

2. Modify Exercise 1 so **draw( )** is a pure virtual function. Try creating an object of type **Shape**. Try to call the pure virtual function inside the constructor and see what happens. Leaving it as a pure virtual, give **draw( )** a definition.
3. Expanding on Exercise 2, create a function that takes a **Shape** object *by value* and try to upcast a derived object in as an argument. See what happens. Fix the function by taking a reference to the **Shape** object.
4. Modify **C14:Combined.cpp** so that **f( )** is **virtual** in the base class. Change **main( )** to perform an upcast and a virtual call.
5. Modify **Instrument3.cpp** by adding a **virtual prepare( )** function. Call **prepare( )** inside **tune( )**.
6. Create an inheritance hierarchy of **Rodent**: **Mouse**, **Gerbil**, **Hamster**, etc. In the base class, provide methods that are common to all **Rodent**s, and redefine these in the derived classes to perform different behaviors depending on the specific type of **Rodent**. Create an array of pointers to **Rodent**, fill it with different specific types of **Rodent**s, and call your base-class methods to see what happens.
7. Modify Exercise 6 so that you use a **vector<Rodent*>** instead of an array of pointers. Make sure that memory is cleaned up properly.
8. Starting with the previous **Rodent** hierarchy, inherit **BlueHamster** from **Hamster** (yes, there is such a thing; I had one when I was a kid), override the base-class methods, and show that the code that calls the base-class methods doesn't need to change in order to accommodate the new type.
9. Starting with the previous **Rodent** hierarchy, add a non virtual destructor, create an object of class **Hamster** using **new**, upcast the pointer to a **Rodent***, and **delete** the pointer to show that it doesn't call all the destructors in the hierarchy. Change the destructor to be **virtual** and demonstrate that the behavior is now correct.
10. Starting with the previous **Rodent** hierarchy, modify **Rodent** so it is a pure abstract base class.
11. Create an air-traffic control system with base-class **Aircraft** and various derived types. Create a **Tower** class with a **vector<Aircraft*>** that sends the appropriate messages to the various aircraft under its control.
12. Create a model of a greenhouse by inheriting various types of **Plant** and building mechanisms into your greenhouse that take care of the plants.
13. In **Early.cpp**, make **Pet** a pure abstract base class.
14. In **AddingVirtuals.cpp**, make all the member functions of **Pet** pure virtuals, but provide a definition for

name( ). Fix **Dog** as necessary, using the base-class definition of **name( )**.

15. Write a small program to show the difference between calling a virtual function inside a normal member function and calling a virtual function inside a constructor. The program should prove that the two calls produce different results.
16. Modify **VirtualsInDestructors.cpp** by inheriting a class from **Derived** and overriding **f( )** and the destructor. In **main( )**, create and upcast an object of your new type, then **delete** it.
17. Take Exercise 16 and add calls to **f( )** in each destructor. Explain what happens.
18. Create a class that has a data member and a derived class that adds another data member. Write a non-member function that takes an object of the base class *by value* and prints out the size of that object using **sizeof**. In **main( )** create an object of the derived class, print out its size, and then call your function. Explain what happens.
19. Create a simple example of a virtual function call and generate assembly output. Locate the assembly code for the virtual call and trace and explain the code.
20. Write a class with one virtual function and one non-virtual function. Inherit a new class, make an object of this class, and upcast to a pointer of the base-class type. Use the **clock( )** function found in **<ctime>** (you'll need to look this up in your local C library guide) to measure the difference between a virtual call and non-virtual call. You'll need to make multiple calls to each function inside your timing loop in order to see the difference.
21. Modify **C14:Order.cpp** by adding a virtual function in the base class of the **CLASS** macro (have it print something) and by making the destructor virtual. Make objects of the various subclasses and upcast them to the base class. Verify that the virtual behavior works and that proper construction and destruction takes place.
22. Write a class with three overloaded virtual functions. Inherit a new class from this and override one of the functions. Create an object of your derived class. Can you call all the base class functions through the derived-class object? Upcast the address of the object to the base. Can you call all three functions through the base? Remove the overridden definition in the derived class. Now can you call all the base class functions through the derived-class object?
23. Modify **VariantReturn.cpp** to show that its behavior works with references as well as pointers.
24. In **Early.cpp**, how can you tell whether the compiler makes the call using early or late binding? Determine

the case for your own compiler.

25. Create a base class containing a **clone( )** function that returns a pointer to a *copy* of the current object. Derive two subclasses that override **clone( )** to return copies of their specific types. In **main( )**, create and upcast objects of your two derived types, then call **clone( )** for each and verify that the cloned copies are the correct subtypes. Experiment with your **clone( )** function so that you return the base type, then try returning the exact derived type. Can you think of situations in which the latter approach is necessary?
26. Modify **OStackTest.cpp** by creating your own class, then multiply-inheriting it with **Object** to create something that can be placed into the **Stack**. Test your class in **main( )**.
27. Add a type called **Tensor** to **OperatorPolymorphism.cpp**.
28. (Intermediate) Create a base **class X** with no data members and no constructor, but with a virtual function. Create a **class Y** that inherits from **X**, but without an explicit constructor. Generate assembly code and examine it to determine if a constructor is created and called for **X**, and if so, what the code does. Explain what you discover. **X** has no default constructor, so why doesn't the compiler complain?
29. (Intermediate) Modify Exercise 28 by writing constructors for both classes so that each constructor calls a virtual function. Generate assembly code. Determine where the VPTR is being assigned inside each constructor. Is the virtual mechanism being used by your compiler inside the constructor? Establish why the local version of the function is still being called.
30. (Advanced) If function calls to an object passed by value *weren't* early-bound, a virtual call might access parts that didn't exist. Is this possible? Write some code to force a virtual call, and see if this causes a crash. To explain the behavior, examine what happens when you pass an object by value.
31. (Advanced) Find out exactly how much more time is required for a virtual function call by going to your processor's assembly-language information or other technical manual and finding out the number of clock states required for a simple call versus the number required for the virtual function instructions.
32. Determine the **sizeof** the VPTR for your implementation. Now multiply-inherit two classes that contain virtual functions. Did you get one VPTR or two in the derived class?
33. Create a class with data members and virtual functions. Write a function that looks at the memory in an object of your class and prints out the various pieces of it. To do this you will need to experiment and iteratively discover where the VPTR is located in the object.
34. Pretend that virtual functions don't exist, and modify **Instrument4.cpp** so that it uses **dynamic_cast** to make the equivalent of the virtual calls. Explain why this is a bad idea.

35. Modify **StaticHierarchyNavigation.cpp** so that instead of using C++ RTTI you create your own RTTI via a virtual function in the base class called **whatAmI( )** and an **enum type { Circles, Squares };**.
36. Start with **PointerToMemberOperator.cpp** from Chapter 12 and show that polymorphism still works with pointers-to-members, even if **operator->*** is overloaded.

---

[54] Compilers may implement virtual behavior any way they want, but the way it's described here is an almost universal approach.

[55] Some compilers might have size issues here but it will be rare.

[56] Smalltalk, Java, and Python, for instance, use this approach with great success.

[57] At Bell Labs, where C++ was invented, there are a *lot* of C programmers. Making them all more efficient, even just a bit, saves the company many millions.

[58] Actually, not all pointers are the same size on all machines. In the context of this discussion, however, they can be considered to be the same.

[Previous Chapter] [Table of Contents] [Index] [Next Chapter]
Last Update:09/27/2001