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 nonvirtual 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.