19. Semi-Persistent Objects

19.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A knowledge of programming in C++.
2. A knowledge of component interface design.
3. A thorough knowledge of the Simple Graphics Editor and its family components.

After completing this chapter you will have:

1. A knowledge of how to safely pass objects from one component to another.
2. A better understanding of how to integrate a number of components into a component family.

19.2. Introduction

The preceding chapters contain several examples of components that use semi-persistent objects. Formally, a semi-persistent object is any object whose lifetime exceeds the life of its creator, but which is never written to persistent storage. Semi-persistent objects exist only in RAM storage, but because they can be passed from one computer to another, their lifetime is unlimited. In our examples, the most extensive use of semi-persistent objects is in the Simple Graphical Editor family of components. Figure 19.1 shows the SGE components that produce and use **CGraphicList** objects. The components outlined with heavy lines are both producers and consumers of **CGraphicList** objects, the others are consumers of **CGraphicList** objects.

In designing this system of components, we have paid little concern to a number of important issues. As a result of ignoring these issues, programs created with these components can be unreliable unless extreme care is taken when passing **CGraphicList** objects between components. The three most important issues that we must address are object ownership, object validation, and the management of virtual functions.

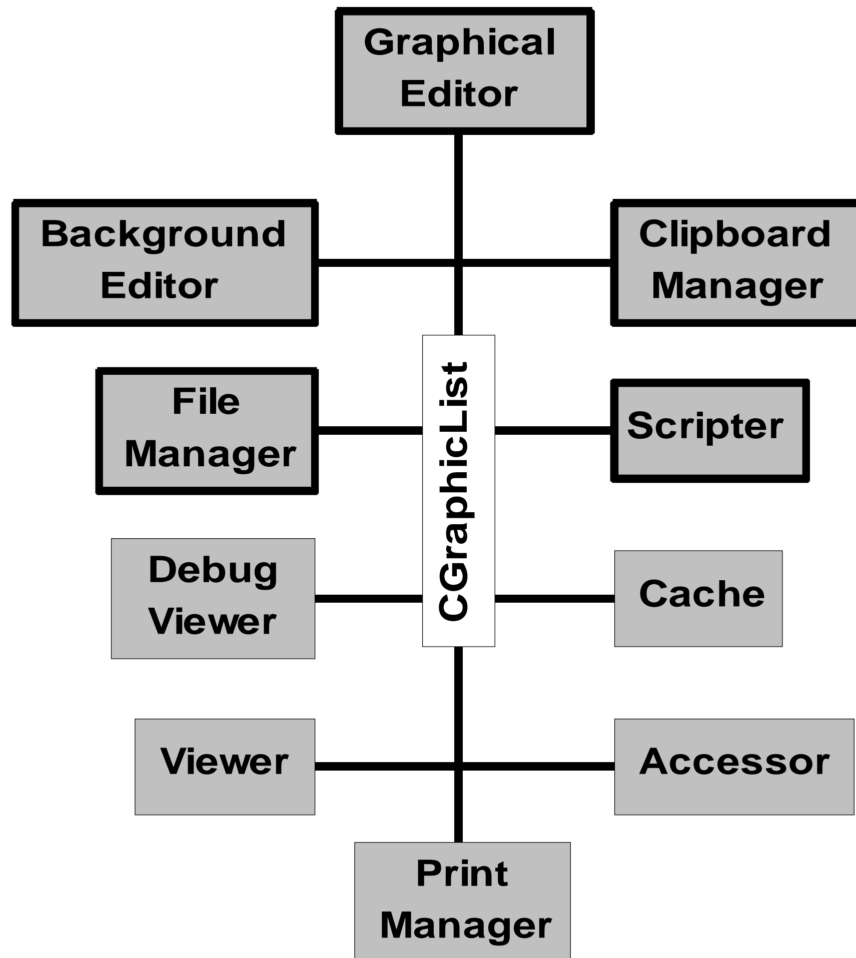

Figure 19.1. The Simple Graphical Editor Components.

19.3. Object Ownership

The issues of object ownership are essentially the same as those for any shared object. The most serious problem occurs when two or more components have pointers to the same object, and one component deletes the object without informing the other. For

example, suppose we are using the Simple Graphical Editor to create **CGraphicList** objects, and the SGE Viewer to display three different views of the objects. This configuration is illustrated in Figure 19.2. If the editor deletes the object, the *OnDraw* routines of the SGE Viewer components will probably fail. The programmer must be careful to call the *ReleaseModel* function for each of the viewers before taking any action that will cause the Graphical Editor to delete its object. Managing semi-persistent is error prone, and would clearly benefit from some formalization of the object-deletion process.

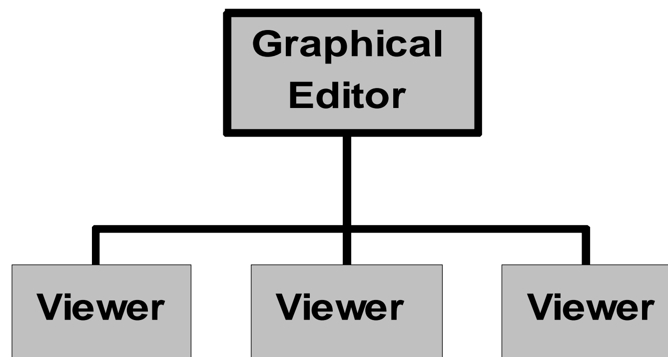

Figure 19.2. Sharing A CGraphicList Object.

One solution to the shared ownership problem illustrated in Figure 19.2 would be to add a reference count to the object and delete the object only when its reference count reaches zero. To guarantee that the reference count is handled properly, a special delete function for **CGraphicList** objects should be created that will decrement the reference count and delete the object only when the count reaches zero. This function should be declared as a friend of the **CGraphicList** class, and the destructor of the **CGraphicList** object could be made private. (Making the destructor private prevents delete statement from being used to delete the object.)

Because the applications we have been dealing with are single-threaded, we can pretty much ignore the problem of object consistency. However if a shared object is used by components on more than one thread, it will be necessary to protect objects with semaphores.

Completely apart from the issues of shared objects, there is the issue of object ownership. For the Integer Sorter described in Chapter 13, objects are passed by value rather than by reference, so the issue of shared objects does not exist. (When a request is made for a **CDynarray** object, the component makes a copy of the object.) If we had decided instead to pass objects by reference, it would be necessary to clearly identify the owner of each object and clearly specify the conditions under which it is permissible to delete an object. Even when passing objects by value, it is necessary for the designer of each component to know whether it is permissible to delete the objects it receives through its properties.

When semi-persistent objects are used to pass data between components, the rules for passing objects must be clearly specified in the design documents before development starts. If objects are to be shared, the owner of the object must be clearly specified, and the rules for deleting the object must be clearly specified. Any operations that delete objects must be clearly identified along with the required operations that must be performed on the other components sharing the object. If an operation deletes an object, it should *always* delete the object, not just under certain conditions. (Here the term *operation* refers to reading a property, writing a property or calling a method.)

19.4. Virtual Functions

Virtual functions represent a thorny problem for semi-persistent objects. If a semi-persistent object (or one of its aggregates) contains a virtual function, you can experience program failures even if you carefully follow the rules of object ownership. The problem is that the object contains a pointer to each virtual function. These pointers are set up by the object constructor, and normally reside within the module containing the constructor. Consider the following scenario. Suppose that the semi-persistent object was developed as a separate project. (Our recommended approach.) Suppose further, that the methods of the object and its aggregates are compiled into a function library that will be included in each component using the persistent object. Under these conditions, *each component will have its own copy of the virtual functions*. In fact, each component will have its own copy of every function it references, but *virtual functions are always referenced*. Let's suppose that a dynamically created component creates a semi-persistent object, passes the object to another component and then relinquishes ownership of the object. If the creator of the object is destroyed, all virtual function pointers may become dangling pointers, *even though the current owner has its own copy of the functions!*

Virtual functions can also cause peculiar problems when debugging a set of components. Suppose a virtual function F is used only by component X, but component X receives all its objects from component Y. If you find a bug in function F, it will manifest itself during the execution of component X. However fixing function F and recompiling X will not make the problem go away. The object was created by component Y, therefore the execution of function F will use the copy in component Y not the copy in

component X. In fact, it's not necessary to recompile X at all. The only thing you need to do is recompile Y. (You can spend *hours* figuring this out on your own.)

One way to avoid this difficulty is to avoid using compile-time libraries for semi-persistent objects. Instead, create an independent load library for the component, and use the library for each component you create. (In the windows operating system these are known as DLLs.) This will give you a single copy of the virtual functions that will be used by all components, but it will complicate the deployment of your components, because each component that uses the semi-persistent object must have access to the run-time library.

19.5. Object Validation

As yet, we have not addressed the problem of receiving an invalid address for a semi-persistent object. In all of our examples, we have naively assumed that every address we receive is valid. Naturally, this can have disastrous consequences, but the problem of object validation is not straightforward. At the very least it is necessary to observe the following conventions.

Every property that is used to pass addresses to a component must be *run-time only*. If not, it is inevitable that some programmer using the component will attempt to assign a value to the property at design time, which will probably cause immediate failure, not just of the component, but of the entire development system. It is also inevitable that this will happen only after the programmer has completed six hours of changes without saving.

No property that is used to pass addresses should be persistent. Addresses are not valid from one invocation to the next, and attempting to restore an address from

persistent storage will probably make the module containing the component completely inaccessible. (The development system will fail every time the module is accessed.)

One method of performing a sanity check on an object address is to require all semi-persistent objects to contain a class signature identifying the class to which they belong. Naturally we would want this signature to be at the beginning of the object, just in case we get the address of an invalid object that is much shorter than the one we were expecting. Unfortunately, in C++ it is difficult to be certain of the location of data items within an object. Visual C++ guarantees the order of data items *only between public/protected/private labels*. If an object contains both public and a private variables, it is difficult to determine which comes first in the object, and the relative order is not guaranteed to be consistent between different compilers or different versions of the same compiler. As a practical matter, most compiler writers want function libraries to be consistent over versions and they also want their function libraries to be usable with other products, so the problem is not as bad as it might first appear. We recommend placing a private signature variable at the beginning of the class description. This variable should be of type long (no object is shorter than 4 bytes), and should contain a “magic number” that identifies the class of the object. If you are using a large number of semi-persistent objects, make sure each magic number is unique. The value of the magic number should be tested using a non-virtual member function.

19.6. Conclusion

The main problem with using semi-persistent objects is that virtually all programming practices associated with memory-based objects, assume that they are ephemeral and contained within a single invocation of a single program. While it is true that this view is

changing rapidly, the problems discussed in the preceding section suggest that there is still much work to be done. Indeed, the mere fact that we must coerce object pointers to long integers suggests that we are straining the design parameters of the underlying technology.

While it is impossible to predict the future, we can at least suggest some innovations that would facilitate the use of semi-persistent objects. We find that the current state of the art of compiler technology to be insufficient to support semi-persistent objects. In the functional view of programming, all functions belong to a program or to a module. This leads to the problems with virtual functions described in the preceding section. The use of dynamic link libraries to bypass the problem is only a stop-gap measure. The real problem is that the functions of an object, especially the virtual functions, must be treated as part of the object, and independent of a particular module. When an object is instantiated, its functions must be instantiated in a way that is independent of the module creating the object. This would enable the object to have an existence that is independent of its creator.

It would be helpful to have some operating system support for object validation. It would be nice to be able to ask the operating system, “Does this address point to an X?” and receive an accurate reply. Ideally, every object that is created anywhere should have its own unique address. When the object is destroyed, the address would be discarded and never reused. Of course, this would require extremely wide addresses, probably on the order of 200 bits or more, but such a practice would simplify the sharing of objects between address spaces and between different machines. Existing technology is quite capable of copying complex objects between address spaces and even between different

machines, but this requires changing pointers into a machine-independent form, and reconstituting them when the object reaches its final destination. (Transferring functions from one machine to another is another matter.)

These issues are the subject of intense ongoing research both in academia and in the commercial world. It will be interesting to see what the future brings.