CHAPTER

1 COM as a Better C++

```
template <class T, class Ex>
class list_t : virtual protected CPrivateAlloc {
  list<T**> m_list;
  mutable TWnd m_wnd;
  virtual ~list_t(void);
protected:
  explicit list_t(int nElems, ...);
  inline operator unsigned int *(void) const
  { return reinterpret_cast<int*>(this); }
  template <class X> void clear(X& rx) const throw(Ex);
};
```

Anonymous, 1996

C++ has been with us for some time now. The community of C++ programmers is quite large, and much is known about the traps and pitfalls of the language. From its inception, C++ has benefited from a highly literate team of developers who, while working at Bell Laboratories, not only produced the first C++ development product (CFRONT) but also published many of the seminal works on C++. Most of the C++ canon was published in the late 1980s and early 1990s. During this era, many C++ developers (including the authors of virtually every important C++ book) worked on UNIX workstations and built fairly monolithic applications using the compiler and linker technology of the day. The environment in which this generation of developers worked has understandably shaped much of the mindset of the C++ community at large.

One of the principal goals for C++ was to allow programmers to build user-defined types (UDTs) that could be reused outside their original implementation context. This principle underpins the idea of class libraries or frameworks as we know them today. Since the introduction of C++, a marketplace for C++ class libraries emerged, albeit rather slowly. One reason that this marketplace has not grown as fast as one might expect is related to the

NIH (not-invented-here) factor among C++ developers. Often the notion of reusing another developer's code seems like more effort than simply reimplementing from scratch. Sometimes this perception is due to sheer hubris on the part of the developer. Other times, the resistance to reuse is based on the mental overhead required to understand someone else's design and programming style. This is especially true for wrapper-style libraries, where one needs to understand not only the underlying technology being wrapped but also the additional abstractions added by the library itself.

To exacerbate the problem, many libraries are documented in a manner that assumes the user of the library will refer to the library's source code as the ultimate reference. This sort of white box reuse often results in a tremendous amount of coupling between the client application and class library, adding to the fragility of the overall code base over time. The coupling effect reduces the modularity of a class library and makes it more difficult to adapt to changes in the underlying library implementation. This encourages clients to treat the library as yet another part of the project's source code base and not as a modular reusable component. In fact, developers have actually been known to modify commercial class library sources to fit their own needs, producing a "private build" that is better suited to the programmer's application but no longer really the original library.

Reuse has always been one of the classic motivations for object orientation. Despite this fact, writing C++ classes that are easily reused is fairly difficult. Beyond the design-time and development-time obstacles to reuse that are part of the C++ culture, there are also a fair number of runtime obstacles that make the C++ object model a less than ideal substrate for building reusable software components. Many of these obstacles stem from the compilation and linkage model assumed by C++. This chapter will look at the technical challenges to treating C++ classes as reusable components. Each challenge will be addressed by presenting programming techniques based on currently available off-the-shelf technology. Through disciplined application of these techniques, this chapter presents an architecture for component reuse that allows dynamic and efficient composition of systems from independently developed binary components.

Software Distribution and C++

To understand the problems related to using C++ as a component substrate, it helps to examine how C++ libraries were distributed in the late 1980s. Consider a library vendor who has developed an algorithm that can perform substring searches on O(1) time (i.e., the search time will be constant and not proportional to the length of the target string). This is a nontrivial task, admittedly. To make the algorithm as simple to use as possible, the vendor would create a string class based on the algorithm that would represent fast text strings in any client program. To do this, the vendor would prepare a header file that contains a class definition:

```
// faststring.h //////////////////////////////
class FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

With the class definition in place, the vendor would implement the member functions in a separate source file:

```
// faststring.cpp ///////////////////////////////////
#include "faststring.h"
#include <string.h>
FastString::FastString(const char *psz)
: m_psz(new char[strlen(psz) + 1]) {
  strcpy(m_psz, psz);
}

FastString::~FastString(void) {
  delete[] m_psz;
}

int FastString::Length(void) const {
  return strlen(m_psz);
}

int FastString::Find(const char *psz) const {
  // O(1) lookup code deleted for clarity[1]
}
```

[1] At the time of this writing, the author did not have a working implementation of this algorithm suitable for publication. The details of such an implementation are left as an exercise for the reader.

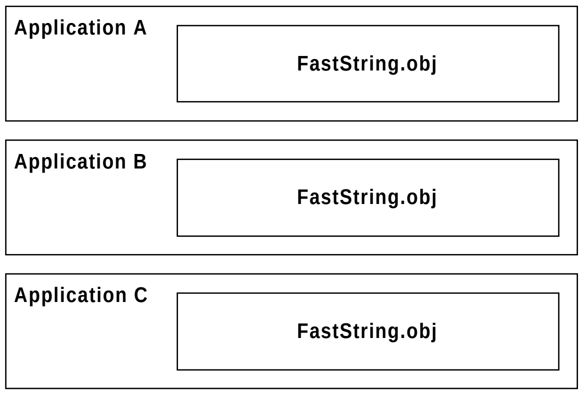

Figure 1.1 Three `FastString` Clients

Traditionally, C++ libraries have been distributed in source code form. The users of a library would be expected to add the implementation source files to their make system and recompile the library sources locally using their C++ compiler. Assuming that the library adhered to a commonly supported subset of the C++ programming language, this was a perfectly workable approach. The net effect of this procedure was that the executable code of the library would be bundled as part of the overall client application.

Assume that for the `FastString` class just shown, the generated machine code for the four methods occupied 16MB worth of space in the target executable image (remember, to perform O(1) search, a lot of code might be needed given the standard time versus space trade-off that binds most algorithms). As shown in Figure 1.1, if three applications use the `FastString` library, each of the three executables will contain the 16MB worth of code. This means that if an end-user installs all three client applications, the `FastString` implementation occupies 48MB worth of disk space. Worse yet, if the end-user runs the three client applications simultaneously, the `FastString` code occupies 48MB worth of virtual memory, as the operating system cannot detect the duplicate code that is present in each executable image.

One additional problem with this scenario is that once the library vendor finds a defect in the `FastString` class, there is no way to field-replace the implementation. Once the `FastString` code is linked into the client application, one can no longer simply replace the `FastString` code directly at the end-user's machine. Instead, the library vendor must broadcast source code updates to the developer of each client application and hope that they will rebuild their applications to take advantage of the repairs that were made. Clearly, the modularity of the `FastString` component is lost once the client runs the linker and produces the final executable.

Dynamic Linking and C++

One technique for solving the problems just stated is to package the FastString class as a Dynamic Link Library (DLL). This can be done in a variety of ways. The simplest technique is to use a class-level compiler directive to force all of the methods of FastString to be exported from the DLL. The Microsoft C++ compiler provides the __declspec(dllexport) keyword for just this purpose:

```
class __declspec(dllexport) FastString {
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

When this technique is used, all of the methods of FastString will be added to the exports list of the FastString DLL, allowing runtime resolution of each method name to its address in memory. In addition, the linker will produce an import library that exposes the symbols for FastString's methods. Instead of containing the actual code, the import library simply contains references to the file name of the DLL and the names of the exported symbols. When the client links against the import library, stubs are added to the executable that inform the loader at runtime to load the FastString DLL dynamically and resolve any imported symbols to their corresponding locations in memory. This resolution happens transparently when the client program is started by the operating system.

Figure 1.2 illustrates the runtime model of FastString when exposed from a DLL. Note that the import library is fairly small (roughly twice the combined size of exported symbol text). When the class is exported from a DLL, the FastString machine code needs to exist only once on the user's hard disk. When multiple clients access the code for the library, the operating system's loader is smart enough to share the physical memory pages containing FastString's read-only executable code between all client programs. In addition, if the library vendor finds a defect in the source code, it is theoretically possible to ship a new DLL to the end-user, repairing the defective implementation for all client applications en masse. Clearly, moving the FastString library into a DLL is an important step toward turning the original C++ class into a field-replaceable and efficient reusable component.

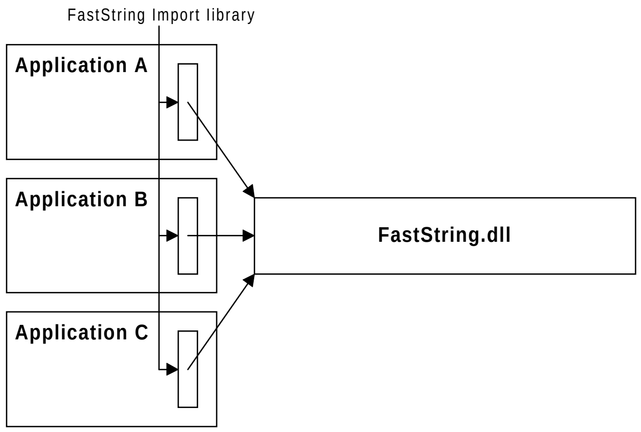

Figure 1.2 `FastString` as a DLL

C++ and Portability

Once the decision is made to distribute a C++ class as a DLL, one is faced with one of the fundamental weaknesses of C++, that is, lack of standardization at the binary level. Although the ISO/ANSI C++ Draft Working Paper attempts to codify which programs will compile and what the semantic effects of running them will be, it makes no attempt to standardize the binary runtime model of C++. The first time this problem will become evident is when a client tries to link against the `FastString` DLL's import library from a C++ development environment other than the one used to build the `FastString` DLL.

To allow operator and function overloading, C++ compilers typically mangle the symbolic name of each entry point to allow many uses of the same name (either with different argument types or in different scopes) without breaking existing C-based linkers. This technique is often referred to as name mangling. Despite the fact that the C++ Annotated Reference Manual (ARM) documented the encoding scheme used by CFRONT, many compiler vendors have elected to invent their own proprietary mangling scheme. Because the `FastString` import library and DLL export symbols using the mangling scheme of the compiler that built the DLL (e.g., GNU C++), clients that are compiled using a different compiler (e.g., Borland C++) will not be able to link successfully with the import library. The classic technique of using `extern "C"` to disable symbolic mangling would not help in this case, as the DLL is exporting member functions, not global functions.

One technique that could alleviate this problem is to play tricks on the client's linker by using a Module Definition File (commonly known as a DEF file). One feature of DEF files is that they allow exported symbols to be aliased to different imported symbols. Given enough time and information about each compiler's mangling scheme, the library vendor can produce a custom import library for each compiler. While tedious, this allows any compiler to gain link-level compatibility with the DLL, provided the library vendor has anticipated its use ahead of time and has provided a proper DEF file.

Once one overcomes the issues related to linking, one now has to deal with the far more problematic area of incompatibilities related to the generated code. For all but the simplest language constructs, compiler vendors often elect to implement language features in proprietary ways that render objects "untouchable" by code generated by any other compiler. Exceptions are a classic example of such a language feature. A C++ exception thrown from a function compiled with the Microsoft compiler cannot be caught reliably by a client program compiled with the Watcom compiler. This is because the DWP does not mandate what a language feature must look like at runtime, so it is perfectly legal for each compiler vendor to implement a language feature in a unique and innovative manner. For building a stand-alone single-binary executable, this is a nonissue, as all of the code will be compiled and linked using a common development environment. For building multibinary component-based executables this is a tremendous issue, as each component could conceivably be built using a different compiler and linker. The lack of a C++ binary standard limits what language features can be used across DLL boundaries. This means that simply exporting C++ member functions from DLLs is not enough to create a vendor-independent component substrate.

Encapsulation and C++

Assuming that one could overcome the compiler and linker issues outlined in the previous section, the next obstacle to building binary components in C++ is related to encapsulation. Consider what would happen if an organization that was using `FastString` in an application managed to accomplish the unaccomplishable: finishing development and testing two months prior to a product's ship date. Assume that during this two-month period, some of the more skeptical developers decide to test `FastString`'s O(1) search algorithm by running a profiler on their application. Much to their surprise, `FastString::Find` would in fact execute extremely quickly irrespective of the presented string length. The `Length` operation, however, would not fare as well. `FastString::Length` uses the C runtime library `strlen` routine, which performs a linear search through the string's data looking for a null terminator. This is an O(n) algorithm that runs slower as the string data gets larger.

Given the fact that the client application may call the Length operation many times, the library vendor will probably be contacted by the once-skeptical developer and asked to increase the speed of Length to operate in constant time as well. There is one catch. The developer has completed development and may not be willing to change one line of source code to take advantage of the newly enhanced Length method. Also, several other vendors may already have shipped products based on the current version of FastString, so the library vendor is morally bound not to break these applications along the way.

At this point, one simply needs to look at the class definition for FastString and decide what can change and what must remain constant in order to keep the installed base happy. Fortunately, FastString was designed with encapsulation in mind, and all of its data members are private. This should afford a great deal of flexibility, as no client programs can write code that directly accesses FastString data members. Provided that no changes are made to the four public members of the class, no changes will be required in any client application. Armed with this belief, the library vendor sets off to implement version 2.0 of FastString.

The obvious enhancement is to cache the length of the string in the constructor and return the cached length in a new version of the Length method. As the string cannot be modified after construction, there is no need to worry about recalculating the length multiple times. In fact, the length is already calculated once in the constructor to allocate the buffer, so only a handful of additional machine instructions will be needed. Here is the modified class definition:

```
// faststring.h version 2.0
class __declspec(dllexport) FastString {
  const int m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

Note that the only modification is the addition of a private data member. To initialize this member properly, the constructor would need to be modified as follows:

```
FastString::FastString(const char *psz)
: m_cch(strlen(psz)), m_psz(new char[m_cch + 1]) {
  strcpy(m_psz, psz);
}
```

Given this cached length, the Length method now becomes trivial:

```
int FastString::Length(void) const {
  return m_cch; // return cached length
}
```

With these three modifications in place, the library vendor can now rebuild the FastString DLL and the corresponding test harness that fully exercises every aspect of the FastString class. The vendor would be pleasantly surprised that the principle of encapsulation payed off and that no source-level modifications were required in the test harness. Once the new DLL has been verified to work properly, the library vendor ships version 2.0 of FastString to the client, confident that all work is finished.

When the clients who contracted the changes receive the updated FastString, they integrate the new class definition and DLL into their source code control system and fire off a build to test the new and improved FastString. Like the library vendor, they too are pleasantly surprised, as no source code modifications are required to take advantage of the new version of Length. Encouraged by this experience, the development team convinces management to add the new DLL to the final "golden master" CD that is about to go to press. In a rare occurrence, management caves in to the wishes of the enthusiastic developers and adds the new DLL to the final product. Like most installation programs, the setup script for the client's product is designed to silently overwrite any older versions of the FastString DLL that may be present on the end-user's machine. This seems innocent enough, as the modifications did not affect the public interface of the class, so the silent machine-wide upgrading to FastString version 2.0 should only enhance any existing client applications that had been previously installed.

Imagine the following scenario: End-users finally receive their copies of the client's highly anticipated product. Each end-user immediately drops everything and installs the new application on their machine to try it out. After the thrill of being able to perform extremely fast text searches wears off, the end-user goes back to his or her normal life and launches a previously installed application that also happens to use the FastString DLL. For the first few minutes all is well. Then, suddenly, a dialog appears indicating that an exception has occurred and that all of the end user's work has been lost. The end-user tries to launch the application again and this time gets the exception dialog almost immediately. The end-user, accustomed to using modern commercial

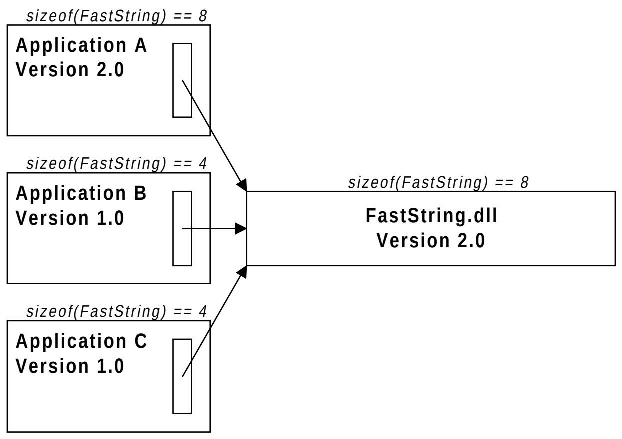

Figure 1.3 C++ and Encapsulation

software, reinstalls the operating system and all applications, but even this does not keep the exception from reoccurring. What happened?

What happened was that the library vendor was lulled into believing that C++ supported encapsulation. Whereas C++ does support syntactic encapsulation via its private and protected keywords, the C++ draft standard has no notion of binary encapsulation. This is because the compilation model of C++ requires the client's compiler to have access to all information regarding object layout in order to instantiate an instance of a class or to make nonvirtual method calls. This includes information about the size and order of the object's private and protected data members. Consider the scenario shown in Figure 1.3. Version 1.0 of `FastString` required four bytes per instance (assuming `sizeof(char *) == 4`). Clients written to version 1.0 of the class definition allocate four bytes of memory to pass to the class's constructor. Version 2.0 of the constructor, destructor, and methods (which are the versions present in the DLL on the end-user's machine) all assume that the client has allocated eight bytes per instance (assuming `sizeof(int) == 4`) and have no reservations about writing to all eight bytes. Unfortunately, in version 1.0 clients, the second four bytes of the object really belong to someone else and writing a pointer to a text string at this location is considered rude, as the exception dialog indicates.

One common solution to the versioning problem is to rename the DLL each time a new version is produced. This is the strategy taken by the Microsoft Foundation Classes (MFC). When the version number is encoded into the DLL's file name (e.g., `FastString10.DLL`, `FastString20.DLL`), clients always load the version of the DLL that they were built against, irrespective of what other versions may be present on the system. Unfortunately, over time, the number of versioned DLLs present on the end-user's system could conceivably exceed the number of actual client applications due to poor software configuration practices. Simply examining the system directory of any computer that has been in use for more than six months would reinforce this.

Ultimately, this versioning problem is rooted in the compilation model of C++, which was not designed to support independent binary components. By requiring client knowledge of object layout, C++ introduces a tight binary coupling between the client and object executables. Normally, binary coupling works to C++'s favor, as it allows compilers to produce extremely efficient code. Unfortunately, this tight binary coupling prevents class implementations from being replaced without client recompilation. Because of this coupling and the compiler and linker incompatibilities mentioned in the previous section, simply exporting C++ class definitions from DLLs does not provide a reasonable binary component architecture.

Separating Interface from Implementation

The concept of encapsulation is based on separating what an object looks like (its interface) from how it actually works (its implementation). The problem with C++ is that this principle does not apply at a binary level, as a C++ class is both interface and implementation simultaneously. This weakness can be addressed by modeling the two abstractions as two distinct entities and C++ classes. By defining one C++ class to represent the interface to a data type and another C++ class as the data type's actual implementation, the object implementor can theoretically modify the implementation class's details while holding the interface class constant. All that is needed is a way to associate the interface with its implementation without revealing any implementation details to the client.

The interface class should describe only what the implementor wants the client to think the underlying data type looks like. Since the interface should not betray any implementation details, the C++ interface class should not contain any of the data members that will be used in the implementation of the object. Instead, the interface class should contain only method declarations for each public operation of the object. The C++ implementation class will contain the actual data members required to implement the object's functionality. One simple approach would be to use a handle class as the interface. The handle class would simply contain an opaque pointer whose type would never be fully defined in the client's scope. The following class definition demonstrates this technique:

```
// faststringitf.h
class __declspec(dllexport) FastStringItf {
  class FastString; // introduce name of impl. class
  FastString *m_pThis; // opaque pointer
                       // (size remains constant)
public:
  FastStringItf(const char *psz);
  ~FastStringItf(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

Note that the binary layout of this interface class does not change as data members are added to or removed from the implementation class, FastString. Also, the use of forward declaration means that the class definition for FastString is not needed for this header file to compile. This effectively hides all details of FastString's implementation from the client's compiler. When using this technique, the machine code for the interface methods becomes the only entry point into the object's DLL and their binary signatures will never change. The implementations of the interface class's methods simply forward the method calls on to the actual implementation class:

```
// faststringitf.cpp/// (part of DLL, not client) //
#include "faststring.h"
#include "faststringitf.h"
FastStringItf::FastStringItf(const char *psz)
: m_pThis(new FastString(psz)) {
  assert(m_pThis != 0);
}

FastStringItf::~FastStringItf(void) {
  delete m_pThis;
}

int FastStringItf::LengthItf(void) const {
  return m_pThis->Length();
}

int FastStringItf::Find(const char *psz) const {
  return m_pThis->Find(psz);
}
```

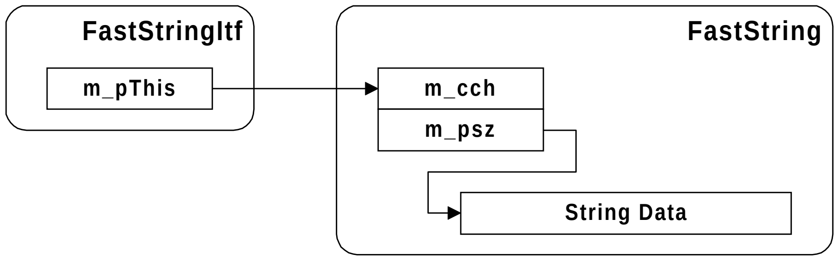

Figure 1.4 Handle Classes as Interfaces

These forwarding methods would be compiled as part of the FastString DLL, so as the layout of the C++ implementation class FastString changes, the call to the new operator in FastStringItf's constructor will be recompiled simultaneously, ensuring that enough memory is always allocated. Again, the client never includes the class definition of the C++ implementation class FastString. This affords the FastString implementor the flexibility to evolve the implementation over time without breaking existing clients.

Figure 1.4 shows the runtime model of using handle classes to separate interface from implementation. Note that the level of indirection introduced by the interface class imposes a binary firewall between the client and the object implementation. This binary firewall is a very precise contract describing how the client can communicate with the implementation. All client-object communications take place through the interface class, which imposes a very simple binary protocol for entering the domain of the object's implementation. This protocol does not rely on any details of the C++ implementation class.

Although the approach of using handle classes has its merits and certainly brings us closer to being able to safely expose a class from a DLL, it also has its weaknesses. Note that the interface class needs to forward each method call to the implementation class explicitly. For a simple class like FastString with only two public operations, a constructor and a destructor, this is not a burden. For a large class library with hundreds or thousands of methods, writing these forwarding routines would become quite tedious and potentially error prone. Also, for performance-critical domains, the cost of making two function calls for each method (one call to the interface, one nested call to the implementation) is less than ideal. Finally, the handle class technique does not completely address the problems of compiler/linker compatibility, which ultimately must be solved if we are to have a truly usable substrate for reusable components.

Abstract Bases as Binary Interfaces

It turns out that applying the technique of separation of interface from implementation can solve C++'s compiler/linker compatibility problems as well; however, the definition of the interface class must take a somewhat different form. As noted previously, the compatibility problems stem from various compilers having different ideas of (1) how to represent language features at runtime and (2) how symbolic names will be represented at link time. If one were to devise a technique for hiding compiler/linker implementation details behind some sort of binary interface, this would make a C++-based DLL usable by a much larger audience.

Ensuring that the binary firewall imposed by the C++ interface class uses no compiler-variant language features can solve the problem of compiler/linker dependence. To accomplish this independence, one must first identify the aspects of the language that have uniform implementations across compilers. Certainly, the runtime representation of composite types such as C-style structs can be held invariant across compilers. This is a basic assumption that any C-based system-call interface must make, and it is sometimes achieved by using conditionally compiled type definitions, pragmas, or other compiler directives. The second assumption that can be made is that all compilers can be coerced to pass function parameters in the same order (right to left, left to right) and that stack cleanup can be done in a uniform manner. Like struct compatibility, this is also a solved problem that often requires conditionally compiled compiler directives to enforce a uniform stack discipline. The WINAPI/WINBASEAPI macros from the Win32 API are an example of this technique. Each function exposed from the system DLLs is defined with these macros:

```
WINBASEAPI void WINAPI Sleep(DWORD dwMsecs);
```

Each compiler vendor defines these preprocessor symbols to produce compliant stack frames. Although in a production environment one would want to use a similar technique for all method definitions, the code fragments in this chapter do not use this technique for the sake of exposition.

The third assumption of compiler invariance is the most critical assumption of all, as it enables the definition of a binary interface: All C++ compilers on a given platform implement the virtual function call mechanism equivalently. In fact, this assumption of uniformity needs to apply only to classes that have no data members and at most one base class that also has no data members. This assumption implies that for the following simple class definition:

```
class calculator {
public:
  virtual void add1(short x);
  virtual void add2(short x, short y);
};
```

all compilers on a given platform must produce equivalent machine code sequences for the following client code fragment:

```
extern calculator *pcalc;
pcalc->add1(1);
pcalc->add2(1, 2);
```

Although the generated machine code does not need to be identical for all compilers, it needs to be equivalent. This means that each compiler must make the same assumptions about how an object of such a class will be represented in memory and how its virtual functions are dynamically invoked at runtime.

It turns out that this is not the tremendous leap of faith that it may seem to be. The runtime implementation of virtual functions in C++ takes the form of vptrs and vtbls in virtually all production compilers. This technique is based on the compiler silently generating a static array of function pointers for each class that contains virtual functions. This array is called the virtual function table (or vtbl) and contains one function pointer for each virtual function defined in the class or its base class. Each instance of the class contains a single invisible data member called the virtual function pointer (or vptr) that is automatically initialized by the constructor to point to the class's vtbl. When a client calls a virtual function, the compiler generates the code to dereference the vptr, index into the vtbl, and call through the function pointer found at the designated location. This is how polymorphism and dynamic call dispatching are implemented in C++. Figure 1.5 shows the runtime representation of the vptr/vtbl layout for the class calculator shown above.

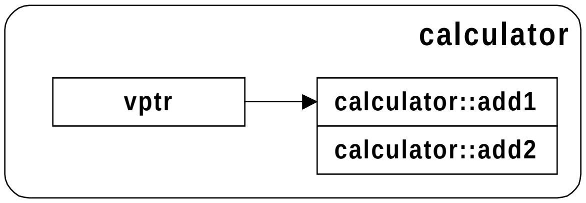

Figure 1.5 vptr/vtbl Layout

Virtually every production-quality C++ compiler currently in use uses the basic concepts of vptrs and vtbls. There are two basic techniques for laying out vtbls: the CFRONT technique and the adjustor thunk technique. Each of these two techniques has its own way of dealing with the subtleties of multiple inheritance. Fortunately, on any given platform, one technique tends to dominate (Win32 compilers use adjustor thunks, Solaris compilers use CFRONT-style vtbls). Fortunately, neither vtbl format affects the C++ source code that one must write but rather is an artifact of the generated code. Consult Stan Lippman's most excellent Inside the C++ Object Model for a great discussion of both techniques.

Based on the assumptions made so far, it is now possible to solve the problem of compiler dependence. Assuming that all compilers on a given platform implement the virtual function call mechanism in the same manner, the C++ interface class can be defined to expose the public operations of the data type as virtual functions, confident that all compilers will generate equivalent machine code for client-side method invocations. This assumption of uniformity requires that no interface class can have data members and no interface class can derive directly from more than one other interface class. Because there are no data members in the interface class, one cannot implement these methods in any sensible manner.

To reinforce this, it is useful to define the interface members as pure virtual functions, indicating that the interface class defines only the potential to call the methods, not their actual implementations:

```
// ifaststring.h ///////////
class IFastString {
public:
  virtual int Length(void) const = 0;
  virtual int Find(const char *psz) const = 0;
};
```

Defining the methods as pure virtual also informs the compiler that no implementation of these methods is required from the interface class. When the compiler generates the vtbl for the interface class, the entry for each pure virtual function will either be null or point to a C runtime routine (_purecall under Microsoft C++) that fires an assertion when called. Had the method definition not been declared as pure virtual, the compiler would have attempted to populate the corresponding vtbl entry with the interface class's method implementation, which of course does not exist. This would result in a link error.

The interface class just defined is an abstract base class. The corresponding C++ implementation class must derive from the interface class and override each of the pure virtual functions with meaningful implementations. This inheritance relationship will result in objects that have an object layout that is a binary superset of the layout of the interface class (which ultimately is just a vptr/vtbl). This is because the "is-a" relationship between derived and base class applies at the binary level in C++ just as it applies at the modeling level in object-oriented design:

```
class FastString : public IFastString {
  const int m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
```

Because `FastString` derives from `IFastString`, the binary layout of `FastString` objects must be a superset of the binary layout of `IFastString`. This means that `FastString` objects will contain a vptr that points to an `IFastString`-compatible vtbl. Since `FastString` is a concrete instantiable data type, its vtbl will contain pointers to actual implementations of the `Length` and `Find` methods. This relationship is illustrated in Figure 1.6.

Even though the public operations of the data type have been hoisted to become pure virtual functions in an interface class, the client cannot instantiate `FastString` objects without having the class definition of the implementation class. Revealing the implementation class definition to the client would bypass the binary encapsulation of the interface, which would defeat the purpose of using an interface class. One reasonable technique for allowing clients to instantiate `FastString` objects would be to export a global function from the

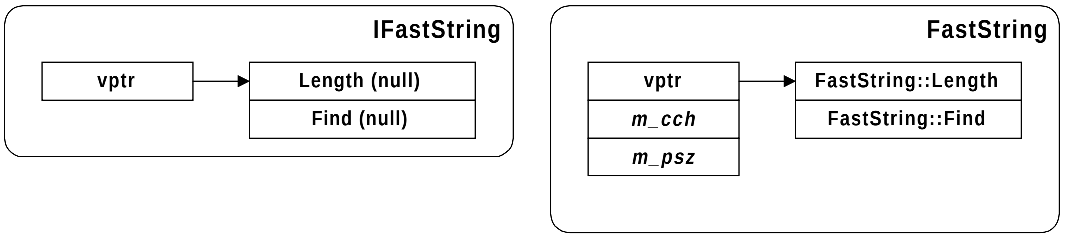

Figure 1.6 Interface/Implementation Classes in Binary

DLL that would call the new operator on behalf of the client. Provided that this routine is exported using extern "C", it would still be accessible from any C++ compiler.

```
// ifaststring.h ///////////
class IFastString {
public:
  virtual int Length(void) const = 0;
  virtual int Find(const char *psz) const = 0;
};

extern "C"
IFastString *CreateFastString(const char *psz);

// faststring.cpp (part of DLL) ///////////
IFastString *CreateFastString (const char *psz) {
  return new FastString(psz);
}
```

As was the case in the handle class approach, the new operator is called exclusively from within the confines of the FastString DLL, which means that the size and layout of the object will be established using the same compiler that compiles all of the implementation's methods.

The last remaining barrier to overcome is related to object destruction. The following client code will compile, but the results are unexpected:

```
int f(void) {
  IFastString *pfs = CreateFastString("Deface me");
  int n = pfs->Find("ace me");
  delete pfs;
  return n;
}
```

The unexpected behavior stems from the fact that the destructor for the interface class is not virtual. This means that the call to the delete operator will not dynamically find the most derived destructor and recursively destroy the object from the outermost type to the base type. Because the FastString destructor is never called, the preceding example leaks the buffer memory used to hold the string "Deface me".

One obvious solution to this problem is to make the destructor virtual in the interface class. Unfortunately, this pollutes the compiler independence of the interface class, as the position of the virtual destructor in the vtbl can vary from compiler to compiler. One workable solution to this problem is to add an explicit Delete method to the interface as another pure virtual function and have the derived class delete itself in its implementation of this method. This results in the correct destructor being executed. The updated version of the interface header file looks like this:

```
// ifaststring.h ///////////
class IFastString {
public:
  virtual void Delete(void) = 0;
  virtual int Length(void) const = 0;
  virtual int Find(const char *psz) const = 0;
};
extern "C"
IFastString *CreateFastString (const char *psz);
```

which implies the corresponding implementation class definition:

```
// faststring.h ////////////////////////////////////
#include "ifaststring.h"
class FastString : public IFastString {
  const int m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
  void Delete(void); // deletes this instance
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
};
// faststring.cpp //////////////////////////////////
#include <string.h>
#include "faststring.h"
IFastString* CreateFastString (const char *psz) {
  return new FastString(psz);
}

FastString::FastString(const char *psz)
: m_cch(strlen(psz)), m_psz(new char[m_cch + 1]) {
  strcpy(m_psz, psz);
}
```

```
void FastString::Delete(void) {
  delete this;
}
FastString::~FastString(void) {
  delete[] m_psz;
}

int FastString::Length(void) const {
  return m_cch;
}

int FastString::Find(const char *psz) const {
  // O(1) lookup code deleted for clarity
}
```

Figure 1.7 shows the runtime layout of FastString.

To use the FastString data type, clients simply need to include the interface definition file and call CreateFastString to begin working:

```
#include "ifaststring.h"

int f(void) {
  int n = -1;
  IFastString *pfs = CreateFastString("Hi Bob!");
  if (pfs) {
    n = pfs->Find("ob");
    pfs->Delete();
  }
  return n;
}
```

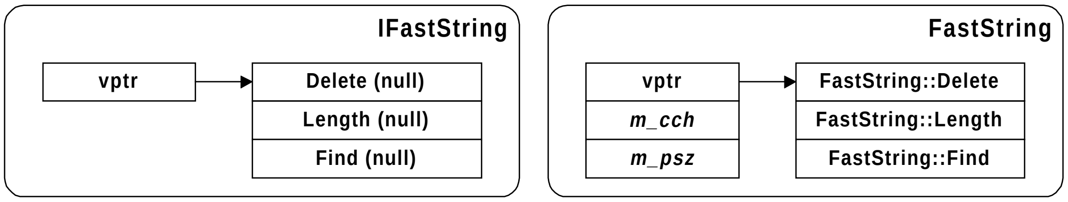

Figure 1.7 FastString Using Abstract Bases as Interfaces

Note that all but one of the entry points into the FastString DLL are virtual functions. The virtual functions of the interface class are always called indirectly via a function pointer stored in a vtbl, without requiring the client to link against their symbolic names at development time. This means that the interface methods are immune to symbolic mangling differences between compilers. The only entry point that is explicitly linked against by name is CreateFastString, the global function that bootstraps the client into the world of FastString. Note, however, that this function was exported using extern "C", which suppresses symbolic mangling. This implies that all C++ compilers expect the import library and DLL to export the same symbol. The net result of using this technique is that one can safely expose a class from a DLL using one C++ environment and access the class from any other C++ development environment. This capability is critical for building a vendor-neutral substrate for reusable components.

Runtime Polymorphism

Deploying class implementations using abstract base classes as interfaces opens up a new world of possibilities in terms of what can happen at runtime. Note that the FastString DLL exports only one symbol, CreateFastString. This makes it fairly trivial for the client to load the DLL dynamically on demand using LoadLibrary and resolve that one entry point using GetProcAddress:

```
IFastString *CallCreateFastString(const char *psz) {
  static IFastString * (*pfn)(const char *) = 0;
  if (!pfn) { // init ptr 1st time through
    const TCHAR szDll[] = __TEXT("FastString.DLL");
    const char  szFn[] = "CreateFastString";
    HINSTANCE h = LoadLibrary(szDll);
    if (h)
      *(FARPROC*)&pfn = GetProcAddress(h, szFn);
  }
  return pfn ? pfn(psz) : 0;
}
```

This technique has several possible applications. One motivation for using the technique is to avoid having operating system (OS)–generated errors when the client program is run on a machine that does not have the object implementation installed. Applications that use optional system components such as WinSock or MAPI use a similar technique to allow an application to run on minimally configured machines. Because the client never needs to link against the DLL's import library, the client has no load dependences on the DLL and can execute on machines that do not have the DLL installed. Another motivation for using this technique might be lazy initialization of the process's address space. Again, because the DLL is not automatically loaded at process initialization time, if the object implementation is never actually used, the DLL is never loaded. Other benefits of using this technique include faster start-up in the client and conservation of address space for long-lived processes that may never actually use the DLL.

Perhaps one of the most interesting applications of this technique is to allow the client to select dynamically between various implementations of the same interface. Given the publicly available definition of the `IFastString` interface, nothing is stopping `FastString`'s original implementor or other third-party implementors from deriving additional implementation classes from the same interface. Like the original implementation class `FastString`, these new implementations will have layouts that are binary compatible with the original interface class. To access these "plug-compatible" implementations, all the client must do is specify the correct filename for the desired implementation's DLL.

To understand how this technique could be applied, assume that the original implementation of `IFastString` performed searches in left-to-right order. This behavior is perfect for languages that parse from left to right (e.g., English, French, German). For languages that parse from right to left, a second implementation of `IFastString` that performs searches using right-to-left order would be more appropriate. This alternative implementation could be built as a second DLL with a distinguished name (e.g., `FastStringRL.DLL`). Assuming both DLLs are installed on an end-user's machine, the client can dynamically select the desired implementation of `IFastString` simply by loading the appropriate DLL at runtime:

```
IFastString *
CallCreateFastString(const char *psz,
                     bool bLeftToRight = true) {
  static IFastString * (*pfnlr)(const char *) = 0;
  static IFastString * (*pfnrl)(const char *) = 0;

  IFastString *(**ppfn)(const char *) = &pfnlr;
  const TCHAR *pszDll = __TEXT("FastString.DLL");

  if (!bLeftToRight) {
    pszDll = __TEXT("FastStringRL.DLL");
```

```
      ppfn = &pfnrl;
    }

    if (!(*ppfn)) { // init ptr 1st time through
      const char  szFn[] = "CreateFastString";
      HINSTANCE h = LoadLibrary(pszDll);
      if (h)
        *(FARPROC*)ppfn = GetProcAddress(h, szFn);
    }
    return (*ppfn) ? (*ppfn)(psz) : 0;
  }
```

When the client calls the function with no second parameter

```
pfs = CallCreateFastString("Hi Bob!");
n = pfs->Find("ob");
```

the original FastString DLL is loaded and the search is performed from left to right. If the client indicates that the string is in a spoken language that parses from right to left

```
pfs = CallCreateFastString("Hi Bob!", false);
n = pfs->Find("ob");
```

the alternative version of the DLL (FastStringRL.DLL) is loaded and the search will be performed starting at the rightmost position of the string. The key observation is that callers of CallCreateFastString do not care which DLL is used to implement the object's methods. All that matters is that a pointer to an IFastString-compatible vptr is returned by the function and that the vptr provides useful and semantically correct functionality. This form of runtime polymorphism is extremely useful for building a dynamically composed system from binary components.

Object Extensibility

The techniques presented so far allow clients to select and load binary components dynamically that can evolve their implementation layout over time without requiring client recompilation. This by itself is extremely useful for building dynamically composable systems. However, one aspect of the object that cannot evolve over time is its interface. This is because the client compiles against the precise signature of the interface class, and any changes to the interface definition require that the client recompile to take advantage of the change. Worse yet, changing the interface definition completely violates the encapsulation of the object (its public interface has changed) and would break all existing clients. Even the most innocuous change, such as changing the semantics of a method while holding its signature constant, renders the installed client base useless. This implies that interfaces are immutable binary and semantic contracts that must never change. This immutability is required to have a stable, predictable runtime environment.

Despite the immutability of interfaces, it is often necessary to expose additional functionality that may not have been anticipated when an interface was initially designed. It is tempting to take advantage of the knowledge of vtbl layout and simply append new methods to the end of an existing interface definition. Consider the initial version of `IFastString`:

```
class IFastString {
public:
  virtual void Delete(void) = 0;
  virtual int Length(void) = 0;
  virtual int Find(const char *psz) = 0;
};
```

Simply modifying the interface class to have additional virtual function declarations after the existing method declarations would result in a binary vtbl format that is a superset of the original version, as any new vtbl entries would appear after those that correspond to the original methods. Object implementations that compile against the new interface definition would have any additional method entries appended to the original vtbl layout:

```
class IFastString {
public:
// faux version 1.0
  virtual void Delete(void) = 0;
  virtual int Length(void) = 0;
  virtual int Find(const char *psz) = 0;
// faux version 2.0
  virtual int FindN(const char *psz, int n) = 0;
};
```

This solution almost works. Clients compiled against the original version of the interface are blissfully ignorant of any vtbl entries beyond the first three. When older clients get newer objects that have the vtbl entry for `FindN`, they continue to work properly. The problem arises when new clients that expect IFastString to have four methods happen to use older objects that do not implement this method. When the client calls the FindN method on an object compiled against the original interface definition, the results are very well defined. The program crashes.

The problem with this technique is that it breaks the encapsulation of the object by modifying the public interface. Just as changing the public interface to a C++ class can cause compile-time errors when the client code is rebuilt, changing the binary interface definition will cause runtime errors when the client code is reexecuted. This implies that interfaces must be immutable and cannot change once published. The solution to this problem is to allow an implementation class to expose more than one interface. This can be achieved either by designing an interface to derive from another related interface or by allowing an implementation class to inherit from several unrelated interface classes. In either case, the client could use C++'s Runtime Type Identification (RTTI) feature to interrogate the object at runtime to ensure that the requested functionality is indeed supported by the object currently in use.

Consider the simple case of an interface extending another interface. To add a FindN operation to IFastString that allows finding the nth occurrence of a substring, one would derive a second interface from IFastString and add the new method definition there:

```
class IFastString2 : public IFastString {
public:
// real version 2.0
  virtual int FindN(const char *psz, int n) = 0;
};
```

Clients can interrogate the object reliably at runtime to discover whether or not the object is IFastString2 compatible by using C++'s dynamic_cast operator:

```
int Find10thBob(IFastString *pfs) {
  IFastString2 *pfs2=dynamic_cast<IFastString2*>(pfs);
  if (pfs2) // the object derives from IFastString2
    return pfs2->FindN("Bob", 10);
  else { // object doesn't derive from IFastString2
    error("Cannot find 10th occurrence of Bob");
    return -1;
  }
}
```

If the object derives from the extended interface, then the dynamic_cast operator returns a pointer to the IFastString2-compliant aspect of the object and the client can call the object's extended method. If the object does not derive from the extended interface, then the dynamic_cast operator will return a null pointer and the client can either choose another implementation technique, log an error message, or just silently continue without the extended operation. This capability of client-defined graceful degradation is critical for building robust dynamic systems that can provide extended functionality over time.

A more interesting scenario arises when new orthogonal functionality needs to be exposed from the object. Consider what it would take to add persistence to the FastString implementation class. Although one could conceivably add Load and Save methods to an extended version of IFastString, it is likely that other types of objects that are not IFastString compatible can also be persistent. Simply creating a new interface that extends IFastString:

```
class IPersistentObject : public IFastString {
public:
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};
```

requires all persistent objects to support the Length and Find operations as well. For some very small subset of objects, this might make sense. However, to make the IPersistentObject interface as generic as possible, it should really be its own interface and not derive from IFastString:

```
class IPersistentObject {
public:
  virtual void Delete(void) = 0;
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};
```

This does not prohibit the FastString implementation from becoming persistent; it simply means that the persistent version of FastString must implement both the IFastString and IPersistentObject interfaces:

```
class FastString : public IFastString,
                   public IPersistentObject {
```

```
  int   m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
// Common methods
  void Delete(void); // deletes this instance
// IFastString methods
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
// IPersistentObject methods
  bool Load(const char *pszFileName);
  bool Save(const char *pszFileName);
};
```

To save a `FastString` to disk, clients can simply use RTTI to bind a pointer to the `IPersistentObject` interface that is exposed by the object:

```
bool SaveString(IFastString *pfs, const char *pszFN){
  bool bResult = false;
  IPersistentObject *ppo =
            dynamic_cast<IPersistentObject*>(pfs);
  if (ppo)
    bResult = ppo->Save(pszFN);
  return bResult;
}
```

This technique works because the compiler has enough knowledge about the layout and type hierarchy of the implementation class to examine an object at runtime to determine whether it in fact derives from `IPersistentObject`. Therein lies the problem.

RTTI is a very compiler-dependent feature. Again, the DWP mandates the syntax and semantics for RTTI, but each compiler vendor's implementation of RTTI is unique and proprietary. This effectively destroys the compiler independence that has been achieved by using abstract base classes as interfaces. This is unacceptable for a vendor-neutral component architecture. One very tractable solution to the problem is to leverage the semantics of `dynamic_cast` without using the actual compiler-dependent language feature. Exposing a well-known method explicitly from each interface that will perform the semantic equivalent of `dynamic_cast` can achieve the desired effect without requiring all entities to use the same C++ compiler:

```
class IPersistentObject {
public:
  virtual void *Dynamic_Cast(const char *pszType) =0;
  virtual void Delete(void) = 0;
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};

class IFastString {
public:
  virtual void *Dynamic_Cast(const char *pszType) =0;
  virtual void Delete(void) = 0;
  virtual int Length(void) = 0;
  virtual int Find(const char *psz) = 0;
};
```

As all interfaces need to expose this method in addition to the `Delete` method already present, it makes a great deal of sense to hoist the common subset of methods to a base interface that all subsequent interfaces could then derive from:

```
class IExtensibleObject {
public:
  virtual void *Dynamic_Cast(const char* pszType) =0;
  virtual void Delete(void) = 0;
};

class IPersistentObject : public IExtensibleObject {
public:
  virtual bool Load(const char *pszFileName) = 0;
  virtual bool Save(const char *pszFileName) = 0;
};

class IFastString : public IExtensibleObject {
public:
  virtual int Length(void) = 0;
  virtual int Find(const char *psz) = 0;
};
```

With this type hierarchy in place, the client is able to query an object dynamically for a given interface using a compiler-independent construct:

```
bool SaveString(IFastString *pfs, const char *pszFN){
  bool bResult = false;
  IPersistentObject *ppo = (IPersistentObject*)
             pfs->Dynamic_Cast("IPersistentObject");
  if (ppo)
    bResult = ppo->Save(pszFN);
  return bResult;
}
```

Assuming the client usage just shown, the required semantics and mechanism for type discovery are in place, but each implementation class must implement this functionality by hand:

```
class FastString : public IFastString,
                   public IPersistentObject {
  int   m_cch; // count of characters
  char *m_psz;
public:
  FastString(const char *psz);
  ~FastString(void);
// IExtensibleObject methods
  void *Dynamic_Cast(const char *pszType);
  void Delete(void); // deletes this instance
// IFastString methods
  int Length(void) const; // returns # of characters
  int Find(const char *psz) const; // returns offset
// IPersistentObject methods
  bool Load(const char *pszFileName);
  bool Save(const char *pszFileName);
};
```

The implementation of Dynamic_Cast needs to simulate the effects of RTTI by navigating the type hierarchy of the object. Figure 1.8 illustrates the type hierarchy for the FastString class just shown. Because the implementation class derives from each interface that it exposes, FastString's implementation of Dynamic_Cast can simply use explicit static casts to limit the scope of the this pointer based on the subtype requested by the client:

```
void *FastString::Dynamic_Cast(const char *pszType) {
  if (strcmp(pszType, "IFastString") == 0)
```

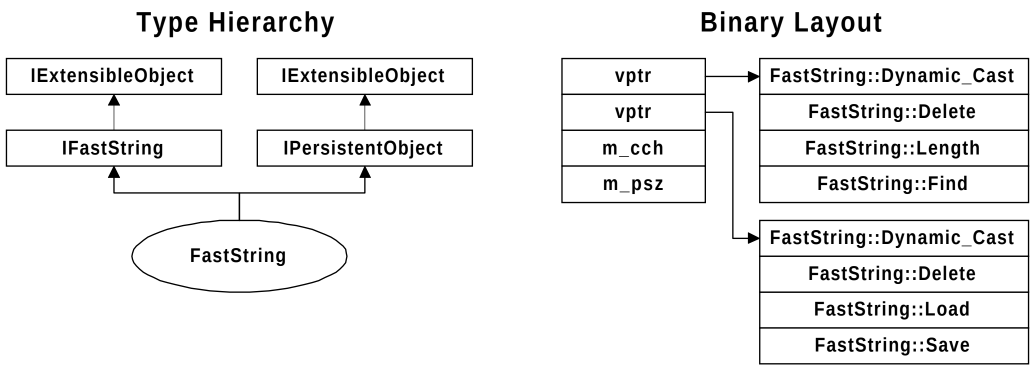

Figure 1.8 FastString Type Hierarchy

```
      return static_cast<IFastString*>(this);
    else if (strcmp(pszType, "IPersistentObject") == 0)
      return static_cast<IPersistentObject*>(this);
    else if (strcmp(pszType, "IExtensibleObject") == 0)
      return static_cast<IFastString*>(this);
    else
      return 0; // request for unsupported interface
  }
```

Because the object derives from the type used in the cast, the compiled versions of the cast statements simply add a fixed offset to the object's this pointer to find the beginning of the base class's layout.

Note that when asked for the common base interface IExtensibleObject, the implementation statically casts itself to IFastString. This is because the intuitive version of the statement

```
return static_cast<IExtensibleObject*>(this);
```

is ambiguous because IFastString and IPersistentObject both derive from IExtensibleObject. If IExtensibleObject was a virtual base class of both IFastString and IPersistentObject, then this cast would not be ambiguous and the statement would compile. However, introducing virtual base classes adds needless runtime complexity to the resultant object and also introduces compiler dependences. This is because virtual bases are yet another C++ language feature that has several proprietary implementations.

Resource Management

One remaining problem with supporting multiple interfaces from a single object becomes clear when examining the client usage pattern of the Dynamic_Cast method. Consider the following client code:

```
void f(void) {
  IFastString *pfs = 0;
  IPersistentObject *ppo = 0;
  pfs = CreateFastString("Feed B0B");
  if (pfs) {
    ppo = (IPersistentObject *)
             pfs->Dynamic_Cast("IPersistentObject");
    if (!ppo)
      pfs->Delete();
    else {
      ppo->Save("C:\\autoexec.bat");
      ppo->Delete();
    }
  }
}
```

Although the object was initially bound via its IFastString interface, the client code is calling the Delete method through the IPersistentObject interface. Given the behavior of multiple inheritance in C++, this is not an issue, as all of the class's IExtensibleObject-derived vtbls will point to the single implementation of the Delete method. However, the client now has to keep track of which pointers are associated with which objects and call Delete only once per object. For the simple code just shown, this is not a tremendous burden. In more complex client code, managing these relationships becomes quite complex and error prone. One way to simplify the client's task is to push the responsibility for managing the lifetime of the object down to the implementation. After all, allowing the client to delete an object explicitly betrays yet another implementation detail: the fact that the object is allocated on the heap.

One simple solution to this problem is to have each object maintain a reference count that is incremented when an interface pointer is duplicated and decremented when an interface pointer is destroyed. This means changing the definition of IExtensibleObject from:

```
class IExtensibleObject {
public:
  virtual void *Dynamic_Cast(const char* pszType) =0;
  virtual void Delete(void) = 0;
};
```

to

```
class IExtensibleObject {
public:
  virtual void *Dynamic_Cast(const char* pszType) =0;
  virtual void DuplicatePointer(void) = 0;
  virtual void DestroyPointer(void) = 0;
};
```

With these methods in place, all users of IExtensibleObject must now adhere to the following two mandates: (1) When an interface pointer is duplicated, a call to DuplicatePointer is required. (2) When an interface pointer is no longer in use, a call to DestroyPointer is required.

These methods could be implemented in each object simply by noting the number of live pointers and destroying the object when no outstanding pointers remain:

```
class FastString : public IFastString,
                   public IPersistentObject {
  int   m_cPtrs; // count of outstanding ptrs
   :      :
public:
// initialize pointer count to zero
  FastString(const char *psz) : m_cPtrs(0) {}
  void DuplicatePointer(void) {
// note duplication of pointer
    ++m_cPtrs;
  }
  void DestroyPointer(void) {
// destroy object when last pointer destroyed
    if (--m_cPtrs == 0)
      delete this;
  }
   :
   :
};
```

This extremely boilerplate code could easily be put into a base class or C pre-processor macro for all implementations to use.

To support these methods, all code that manipulates or manages interface pointers must adhere to the two simple rules of DuplicatePointer/ DestroyPointer. For the implementation of FastString, this means modifying two functions. The CreateFastString function takes the initial pointer that is returned by the C++ new operator and duplicates it onto the stack to return to the client. This implies that a call to DuplicatePointer is required:

```
IFastString* CreateFastString(const char *psz) {
  IFastString *pfsResult = new FastString(psz);
  if (pfsResult)
    pfsResult->DuplicatePointer();
  return pfsResult;
}
```

The other location where the implementation duplicates a pointer is in the Dynamic_Cast method:

```
void *FastString::Dynamic_Cast(const char *pszType) {
  void *pvResult = 0;
  if (strcmp(pszType, "IFastString") == 0)
    pvResult = static_cast<IFastString*>(this);
  else if (strcmp(pszType, "IPersistentObject") == 0)
    pvResult = static_cast<IPersistentObject*>(this);
  else if (strcmp(pszType, "IExtensibleObject") == 0)
    pvResult = static_cast<IFastString*>(this);
  else
    return 0; // request for unsupported interface
// pvResult now contains a duplicated pointer, so
// we must call DuplicatePointer prior to returning
  ((IExtensibleObject*)pvResult)->DuplicatePointer();
  return pvResult;
}
```

With these two modifications in place, the corresponding client code now becomes much more uniform and unambiguous:

```
void f(void) {
  IFastString *pfs = 0;
  IPersistentObject *ppo = 0;
```

```
    pfs = CreateFastString("Feed B0B");
    if (pfs) {
      ppo = (IPersistentObject *)
                 pfs->Dynamic_Cast("IPersistentObject");
      if (ppo) {
        ppo->Save("C:\\autoexec.bat");
        ppo->DestroyPointer();
      }
      pfs->DestroyPointer();
    }
  }
```

Because each pointer is now treated as an autonomous entity with respect to lifetime, the client does not need to be concerned with which pointer refers to which object. Instead, the client simply follows the two simple rules and allows the object to manage its own lifetime. If desired, the idiom of calling `DuplicatePointer` and `DestroyPointer` could easily be hidden behind a C++ smart pointer.

Using this reference counting scheme allows an object to expose multiple interfaces in an extremely uniform manner. The capability to expose multiple interfaces from a single implementation class allows a data type to participate in a variety of contexts. For example, a new persistence subsystem could define its own custom interface for telling objects to load and save themselves to some customized storage medium. The `FastString` class could add support for this functionality simply by deriving from the subsystem's persistence interface. Adding this support does not in any way affect the installed base of clients that may be using the prior persistence interface to save and load a string to disk. Having a mechanism for runtime interface negotiation can act as a cornerstone for building a dynamic system from components that can evolve over time.

Where Are We?

This chapter started with a simple C++ class and examined the issues related to exposing the class as a reusable binary component. The first step was to deploy the class as a Dynamic Link Library (DLL) to decouple the physical package of the class from the packaging of its clients. We then used the notion of interfaces and implementations to encapsulate the implementation details of the data type behind a binary firewall, allowing the object's layout to evolve without requiring client recompilation. When using the abstract base class approach to define interfaces, this firewall took the form of a vptr and vtbl. We then examined techniques for dynamically selecting different polymorphic implementations of a given interface at runtime using LoadLibrary and GetProcAddress. Finally, we used an RTTI-like construct for dynamically interrogating an object to discover whether it in fact implements a desired interface. This construct gave us a technique for extending existing versions of an interface as well as exposing multiple unrelated interfaces from a single object.

In short, we have just engineered the Component Object Model.