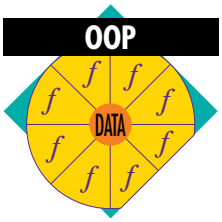


Using SOM with C++

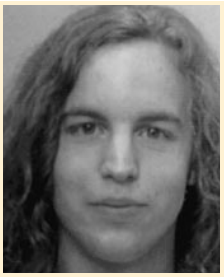
By Jennifer Hamilton, Robert Klarer, Mark Mendell, and Brian Thomson

This article introduces the IBM System Object Model (SOM) and describes how it can be used from C++. It also includes a description of the DirectToSOM (DTS) support provided by some C++ compilers.

A major goal of Object-Oriented Programming (OOP) is to write programs that can be more easily reused and extended than those written using conventional programming practices. The C++ language directly supports OOP techniques through language features such as classes and exception handling (which permit data encapsulation) and templates (a powerful mechanism for realizing generic types).

While there is considerable debate in the industry as to just how successful OOP languages have been with respect to achieving greater software reuse, such discussions typically apply to source code. What about binary reuse? Languages such as C++ not only fail to solve this issue, but actually make it considerably more difficult.

Jennifer Hamilton



Robert Klarer

Object Mapping in C++

The ANSI®/ISO® C++ standardization committee—whose mandate is to standardize the language itself and not the internal implementation of C++ compilers—explicitly chose not to specify a standard object mapping for C++. In fact, *The C++ Annotated Reference Manual* (ARM) suggests several different mapping schemes for handling virtual member function calls through virtual function tables. While most C++ compilers use a variant of this object mapping scheme, there is actually no requirement that the virtual function table scheme be used at all, although alternatives have not generally been used in commercial C++ compilers. Unless the object mapping is exact, the objects cannot be shared. So while the standard may ensure that C++ source code is relatively portable across implementations, it offers programmers no guarantee of the compatibility of binary objects that have been produced using different C++ compilers.

Even the C programming language enjoys an important advantage over C++ in its relative ability to support binary reuse. Most C language subroutine libraries can be exploited in client programs regardless of whether the same compiler, or even the same programming language, was used for both the library and the client code.

As a result, commercial C++ class library vendors cannot ship a single binary to their customers with the expectation that it will be useful regardless of which compiler their customers use. For example, it is common among vendors of class libraries for DOS to ship several binaries with each distribution of their product. Each such binary can be used with only one DOS compiler. Library vendors who want to serve the broadest market possible are thus forced to ship many precompiled versions of their wares to each customer.

Even within an implementation, it is difficult—if not impossible—to make changes to classes so that client code is not affected. For some cases, binary compatibility can be achieved by carefully managing class changes. But changing the size of a C++ class or migrating a member function up the class hierarchy will require recompilation of any clients of that class, including subclasses.

Clearly, we have a problem: there is no guaranteed way to share objects across different C++ implementations, let alone with other OOP languages such as Smalltalk. There is also limited ability to make common updates to classes without requiring recompilation of client code. Now imagine that you are responsible for maintaining a class library that is shared by 47 different applications. The fact that you have lots of shared

code is great, but if you change just one of those shared classes, you may be faced with having to recompile every client application.

System Object Model

System Object Model (SOM), like the conventional C++ object model, is a strategy for mapping instances of class types in memory. However, SOM was intended to support the reuse of binaries in ways that C++ was not. SOM has the advantage of Release-to-Release Binary Compatibility (RRBC), an important feature that breaks the tight dependency between the code that implements a class and the client code that uses it.

RRBC enables you to create and deploy a new version of a class without requiring that you recompile any unmodified application code. For example, you can add function or data members, or even inherit from additional base classes. In general, if you make a change to a SOM class that does not require a source code change in a client, then that client will not need to be recompiled. By packaging your class in a Dynamic Link Library (DLL), you can replace the old DLL with the new one, and all the applications that used it will continue to run without modification.

SOM also offers other features not available in native C++. The language-independent object model allows classes to be implemented in one language, instantiated in a second, and subclassed in a third. SOM supports extensive dynamic facilities, such as querying the properties of objects and classes, and using classes and methods whose names are not known until execution time. SOM also includes Distributed SOM (DSOM), allowing access to objects between processes or even across networks. It is fully compliant with the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) standards.

Experienced C++ programmers may find that the most surprising difference between SOM and the C++ object model is related to the role of object classes. In C++, a class is a syntactic entity that exists only at compile time; it has no representation outside of the source code that defines it. A SOM class, however, is also a SOM object, and always exists at runtime. Because SOM classes are runtime objects, they can provide a number of services to client objects at runtime. For example, each SOM class possesses a method named `somSupportsMethod` that, when invoked with any string, returns a value representing either true or false, depending on whether the class instances support a method whose name matches the parameter string. Other services supported by a SOM class include the following:

- Reporting its name to clients
- Identifying its base classes
- Indicating whether or not a given SOM object is one of its instances
- Reporting the size of its instances
- Reporting the number of methods that its instances support

All interactions with SOM are through standard procedure calls. Because public instance data is not directly supported, any language that supports external calls can use SOM. However, SOM is most heavily used today with C and C++, mainly because of the support for generating bindings for these languages, and the DirectToSOM support provided by some C++ compilers.

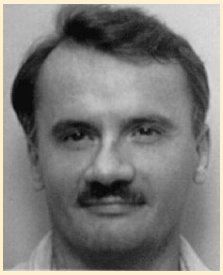
Interface Definition Language

C++ programmers can define SOM classes in one of two ways: either through the CORBA standard Interface Definition Language (IDL), or directly in C++ using a DirectToSOM C++ compiler. IDL is a language-neutral means for describing object interfaces, thereby enabling different compilers and even different programming languages to manipulate shared objects. The IDL definition describes the interface to, but not the implementation of, a SOM class.

The SOM Toolkit includes a compiler that generates bindings for a given target language from the IDL description. Bindings are language-specific macros and procedures that allow a programmer to interact with SOM through a simplified syntax that is more natural for the particular language. For example, the C++ bindings enable SOM objects to be manipulated through C++ pointers to objects, using any C++ compiler. DirectToSOM support, discussed in more detail later in this article, lets the programmer define and use SOM classes directly in C++ without needing IDL definitions.

As an example of C++ bindings, file `hello.idl` in Figure 1 shows the IDL definition for the class `Hello`, with the single method `sayHello`. SOM includes a compiler that will generate usage bindings, implementation bindings, and an implementation template for the class in the files `hello.xh`, `hello.xih`, and `hello.C` respectively.

Mark Mendell



Brian Thomson

hello.idl

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
};
```

hello.C

```
#include "hello.xih"

SOM_Scope void  SOMLINK sayHello(Hello *somSelf,
     Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","sayHello");

}
```

Figure 1. Simple IDL class definition and implementation template

hello.C

```
#include <iostream.h>
#include "hello.xih"

SOM_Scope void  SOMLINK sayHello(Hello *somSelf,
   Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","sayHello");

    cout << "Hello world" << endl;
}
```

main.C

```
#include "hello.xh"

int main(int argc, char *argv[])
{
    Hello *obj;

    obj = new Hello;
    obj->sayHello(somGetGlobalEnvironment());
    delete obj;
    return(0);
}
```

Figure 2. Class implementation template and client code

The usage bindings define the public interface to a SOM class, and are included by clients to create and manipulate objects of that class. The implementation bindings, included by the class implementer, define the class and include private information that is not part of the class interface. Both the implementation and usage bindings files are regenerated completely by the SOM compiler when the class is modified, and should not be updated directly.

The implementation template (file `hello.C` in Figure 2 for the class `Hello`) contains procedure stubs for each method introduced or overridden by the class, and is updated incrementally by the SOM compiler when methods are added or a method signature is modified. The class implementer modifies the implementation template file to define the class behavior.

The first parameter to a method is the target SOM object, from which the address of the instance data can be retrieved by calling `classNameGetData`. The second parameter is an environment parameter that is required for CORBA compliance. In Figure 2 , we have updated the implementation file to make the method `sayHello` print the message "Hello world". The file `main.C` shows a client program that uses the class `Hello`. With the C++ bindings, SOM objects are declared and manipulated as pointers to the given class. You use the `new` operator to create instances of a class. The first time an object of any class is created, the SOM runtime environment will implicitly be initialized, and the first time an instance of a given class is created, the associated class object will be created, along with any parent class objects.

In this example, the second line of the function `main` will initialize the SOM runtime environment, create a class object `Hello`, and allocate storage for an instance of the class `Hello`, which will be assigned to the variable `obj`. The third line invokes the method `sayHello` on the object `obj`, while the final line deallocates the storage for the object. This program can be compiled and run with any C++ compiler. There is much more to the C++ bindings than we have shown here, but this should give you a feel for how SOM classes are defined, implemented, and used through the C++ bindings.

Release Order

The SOM interface does not support direct client access to instance data; all object manipulation is done through procedure calls, so the methods

form the only interface for a class. SOM supports RRBC by maintaining a list of all methods introduced by a class, called the *release order* for the class. Clients use the release order to access methods in the method table for the class. By keeping the order of methods in this list consistent, you can add new methods to the end of the list without forcing recompilation of client code. An IDL modifier is used to specify the release order for a class, listing each method in order by name, so new methods can be defined anywhere and simply added to the end of the list.

The usage bindings for C++ supply functions for each method that uses the release order to map the method invocation to the appropriate slot in the method table for the class.

DirectToSOM C++ Compilers

The capability to generate C++ bindings from an IDL description enables you to create and manipulate SOM objects with any C++ compiler, thereby gaining the advantages of the RRBC support provided by SOM. In addition, those objects can be shared across different C++ implementations or even with different languages such as Smalltalk. In using the C++ bindings, however, you are limited to a subset of the C++ language, which makes migration of existing C++ applications more difficult. You must also use two languages (IDL and C++) to define and manipulate objects.

DirectToSOM (DTS) C++ compilers support and enforce both the C++ and the SOM object models, enabling C++ programmers to take advantage of SOM through C++ language syntax and semantics so that the use of SOM is reasonably transparent and efficient. Instead of first describing SOM classes in IDL, the DTS C++ compiler translates C++ syntax to SOM. You can then have the compiler generate IDL from your C++ declaration, or you may find that you do not need to deal with IDL at all and can work exclusively in DTS C++. Finally, because you write C++ directly, you can use C++ features in your SOM classes that were not available before DTS. These features include templates, operators, constructors with parameters, default parameters, static members, public instance data, and more.

A C++ class is made into a DTS C++ class by inheriting from the class `SOMObject`, which is defined in the header file `<som.hh>`. You can do this explicitly, as shown in Figure 3, or implicitly through compiler switches or pragmas that insert `SOMObject` as a base class. The access

```
#include <som.hh>

class Hello : SOMObject {
  public:
    void sayHello();
};
```

Figure 3. A simple DTS class

specifiers—private, protected, and public—are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs.

Once you have defined a DTS class, what can you do with it? You can create SOM objects statically or dynamically, as simple objects, arrays, or embedded members of other classes (or anywhere else that the declaration of a C++ object is valid). Most C++ rules and syntax apply to DTS classes and objects, with some restrictions. Because the size of a SOM object is not known until runtime, compile-time constant expressions such as `sizeof` are treated as runtime constant expressions. Such operators can still be used with SOM objects, but not in contexts that require compile-time evaluation.

A major inhibitor to RRBC with C++ is the fact that so much information about an object is statically compiled into client code—in particular, the location of instance data and virtual function pointers. Data layout and method calling for a DTS C++ class are performed using the SOM Application Programming Interface (API) instead of the native C++ API. When you run a program defining a DTS C++ class, the compiler will create the corresponding SOM class object at runtime and use it to create and manipulate the object. As a result, unlike a standard C++ object, much of the information about a SOM object and its class—such as the instance size—is not determined until runtime, when the class object is created. This enables class evolution without forcing recompilation of client applications.

The SOM interface does not support direct client access to instance data. C++ instance data members in a DTS class are regrouped into contiguous chunks according to access, in the order of declaration within the class. This regrouping gives efficient access to data members from client code while enabling RRBC. The location of each chunk is determined at runtime. If the declaration order of public and protected data within a class is not changed, and new members are added after any preexisting members of the same

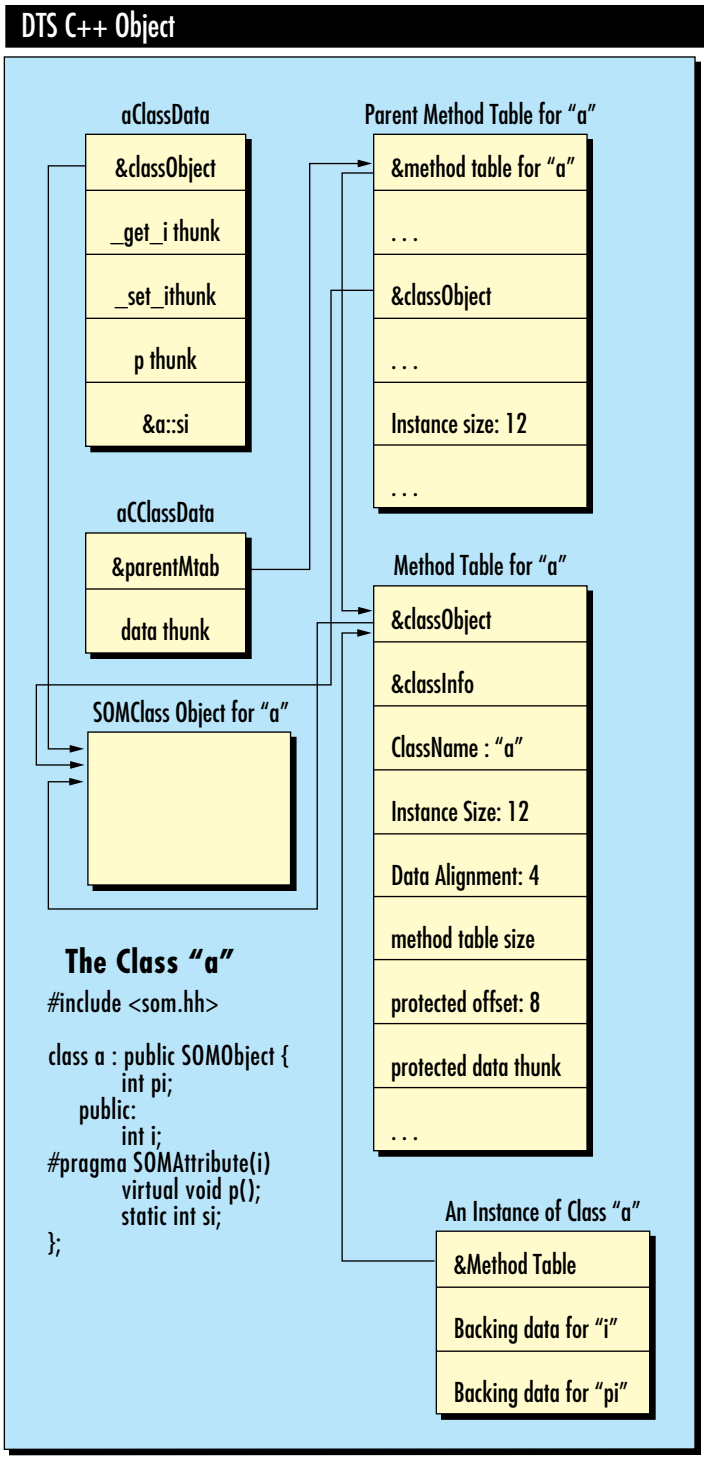


Figure 4. Runtime representation of a DTS C++ object

access, this scheme allows new data members to be added without requiring recompilation of any code outside the class (except for friends).

A DTS class also has a release order that, by default, will contain, in the order of declaration, all member functions and static data members introduced by the class, including those with private and protected access. In general, virtual functions that override virtual functions in a base class do not appear in this list, but will appear in the release order for the introducing class. Using the default, you must add any new member functions or static data members at the end of the class. Instead of relying on declaration order, you can use a pragma to specify the release order. In this case, you can add new release order elements anywhere in the class, but you must add their names to the end of the list.

For DTS classes, instance data and the release order list are created at runtime by SOM and used to manipulate SOM objects, rather than the statically defined compiler constructs used by standard C++. This provides for both RRBC and an implementation-independent object model. As long as the order of list elements does not change and new elements are added to the end of the list, you can add new data members and member functions—including migrating a member function up the class hierarchy—without forcing recompilation of client code.

DTS C++ Object Layout

The layout of a DTS C++ class differs from that of a native C++ class as follows:

- There is only one "method table" pointer in a class. This replaces the native C++ virtual table pointer. In a native C++ class, there may be many virtual table pointers in an object.
- There are no "virtual base pointers" in a SOM class.
- The data members for the SOM class have been reordered to place all the public data members first, then the protected data members, and finally the private data members.
- The order of the data instances for a derived class and that of its base classes is unknown at compile time and must be determined at runtime. This allows a base class to become larger in a new release, and still have the derived classes function correctly. The SOM runtime determines the class layout at startup time.

Figure 4 shows the definition and layout for the DTS C++ class `a`. Associated with every SOM class are two statically defined tables called `xClassData` and `xCClassData` for a given class `x`. A SOM class replaces the native C++ virtual table with the `ClassData` data structure. This table is initialized by the SOM runtime with a pointer to the SOM class object for the class followed by the

addresses of *thunks*, which are small code sequences that branch to the proper virtual function. To call a virtual function, the `ClassData` structure for the class that introduced the virtual function is indexed to get the address of a thunk. This thunk is then called, passing the `this` pointer and any parameters. It is then up to the thunk, which was created by the SOM runtime, to branch to the correct function.

To call the virtual function `p` for class `a` in Figure 4, for example, the function pointer found in `aClassData[3]` is called. The thunk will then call the appropriate method. The same code is used to call a non-virtual or static function. In this case, there is no thunk, but instead the address of the function is contained in the `ClassData` structure. Since the routine to be called never changes, there is no need for any generated code sequences. The address of static data members is also placed in the `ClassData` structure. The `aClassData[4]` structure in Figure 4 contains the address of static member `si`, thereby enabling other languages and implementations to access the data without knowing the external name of the variable.

The `xCClassData` structure contains a pointer to the parent method table for the class, followed by the address of a data thunk. Instance data for a class is accessed by calling the data thunk, which returns the location of the data introduced by the class. The use of the data thunk prevents any dependencies upon relative position or size of introduced data. The parent method table contents include pointers to the method table and class object for the class, along with the instance size. The method table itself contains class-specific information followed by the addresses of the actual methods for the class. Each class instance also contains a pointer to the method table, followed by the instance data for the object. The instance data for class `a` in Figure 4 contains the public data `i` before the private member `pi`, rather than the order of declaration within the class, because of the reordering of instance data by access.

Attributes

Note the `#pragma SOMAttribute(i)` directive for class `a`. A DSOM/CORBA attribute—logical data that is manipulated by `get` and `set` methods—is typically used when working with distributed objects for which direct data access is not possible. You can make a DTS C++ class data member into an attribute using the `SOMAttribute` pragma. The compiler will generate the methods `_get_membername` and `_set_membername` with the same access as the data member, and the actual backing data for the attribute will become private. By default, references to the members outside the class will not have access to the private backing data, so the compiler implicitly transforms member references into calls to the `get` and `set` methods. Within the member functions for the class, the private backing data can be accessed directly.

Differences between Native C++ and DTS C++ Classes

For a DTS C++ class, `sizeof(SOM class)` is a runtime value, not a compile-time value. That is because a SOM class may have a different size each time a program is run, if a shared library that defines a base class is changed to a new version. Within a given execution, the size is fixed. This means that `int a[sizeof(B)];` is not allowed if `B` is a SOM class.

The `offsetof(SOM class, member)` value will give values that are somewhat unusual compared to native C++. The offset of a data member depends on the access mode; public members start at offset 0, but the protected and private members together also start at offset 0. Consider the following class:

```
struct a : SOMObject {
  private:
    int priv;          // offset 4
  protected:
    int prot;          // offset 0
  public:
    int pub;           // offset 0
};
```

This is because the protected and private members are allocated together as a chunk, separate from the public members. In addition, the offset is always relative to the class that introduced the member; `offsetof(derived, base_element)` is *always* equal to `offsetof(base, base_element)`.

For CORBA compliance, SOM class member functions may have an extra hidden parameter (the Environment pointer) as a second parameter after the `this` parameter. Unfortunately, SOM Version 1 was developed before CORBA, and the environment pointer was not present. Older SOM classes do not have an environment pointer, but new ones do. This affects pointer-to-member functions. When calling through a pointer-to-member function, the compiler has to know whether or not to pass an environment pointer.

The compiler assumes that this is determined by the pointer-to-class function and prohibits mismatches. This is generally not a problem for existing C++ code, since all new classes will have the environment pointer.

A further difference lies in the area of inlining. Inlining member functions can inhibit RRBC, so the compiler generates inline functions "out of line." This preserves RRBC, but slows down execution.

Conclusion

SOM provides a great step forward in achieving release-to-release binary compatibility and interlanguage object sharing, with support for both remote and local objects. The C++ bindings generated by the SOM compiler from the IDL description enable SOM objects to be manipulated as C++ objects using any C++ compiler. Going one step further, DirectToSOM C++ compilers allow programmers to take advantage of the power of SOM through standard C++ language constructs and syntax, and to migrate existing C++ classes to SOM. Using SOM with C++, you can take advantage of the C++ support for OOP, using native C++ classes within your implementation and binary-compatible SOM classes for your interface.

SOM provides a great step forward in achieving release-to-release binary compatibility.

References

- Danforth, S. "A Bird's Eye View of SOM." *IBM OS/2 Developer* (Winter 1992).
- Danforth, S., Koenen P., and Tate, B. *Objects for OS/2*. New York, NY: Van Nostrand Reinhold, 1994.
- *C Set ++ Version 3.1 for AIX User's Guide*. (SC09-1968) 1994.
- *SOMObjects Base Toolkit User's Guide Version 2.0*. (SC23-2680) 1993.

Jennifer Hamilton, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Ms. Hamilton has worked in compiler development since joining IBM in 1987, and currently develops the DirectToSOM support for IBM's C Set++ language products. She is the author of two books and numerous articles on programming language-related topics. She has a BSc in Computer Science from the University of Victoria, B.C.

Robert Klarer, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Mr. Klarer has been involved in the development of several of IBM's C++ compilers. Recently, he has contributed to IBM's implementation of DirectToSOM C++. He has a BSc in Computer Science from McMaster University.

Mark Mendell, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Mr. Mendell has worked on compiler development since 1980. He helped develop compilers for Concurrent Euclid, Turing, and Turing Plus while working at the University of Toronto, and has been involved with C++ compiler development since joining IBM in 1991. He has a BSc in Electrical Engineering from Cornell University and a MSc in Computer Science from the University of Toronto.

Brian Thomson, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7. Mr. Thomson has worked in C++ compiler development since joining IBM in 1992. Previously he did research in operating systems, networks, and computer security, as well as languages. He is currently the DirectToSOM architect for IBM's C Set++ language products.