DirectToSOM C++ Overview

By Jennifer Hamilton

DirectToSOM C++ compilers support and enforce both the C++ and the SOM object models. This allows programmers to take advantage of SOM through the C++ language syntax and semantics so that use of SOM is transparent and efficient. This article covers some basic concepts that will help you get started more effectively with DirectToSOM C++.

The System Object Model (SOM) was designed to provide a state-of-the art object model to address problems introduced by object-oriented programming languages: Release-to-Release Binary Compatibility (RRBC) and interlanguage object sharing. SOM provides separation of interface and implementation through a language-independent model, allowing the class client and implementation to be written in different languages and a new version of a class to be supplied without requiring recompilation of unmodified client code.

DirectToSOM C++ compilers support and enforce both the C++ and the SOM object models, allowing C++ programmers to take advantage of SOM through C++ language syntax and semantics so that the use of SOM is reasonably transparent and efficient. DirectToSOM C++ is part of the IBM VisualAge™ C++ and Metaware® High C++ compiler products, and is currently available for AIX, OS/2, MVS®, Windows 95, and Windows NT™. An earlier article ("Using SOM with C++," *AIXpert*, August 1995) provided an overview of SOM and how it is used from C++. This article focuses on DirectToSOM C++ and covers the basic concepts and usage considerations.

Jennifer Hamilton

Defining DirectToSOM C++ Classes

DirectToSOM C++ classes are distinct from native C++ classes. A class is a DirectToSOM class if it inherits from the predefined DirectToSOM class `SOMObject`, defined in the header file `<som.hh>`. This can be achieved in a variety of ways:

- `#pragma SOMAsDefault(on|off|pop)` can be used to instruct the compiler to implicitly insert `SOMObject` as the base class for all classes while `on` is in effect. For example, in Figure 1, class A is a DirectToSOM class, while class B is a native C++ class. (`pop` returns to the mode before the most recent `SOMAsDefault pragma` occurrence.) The `SOMAsDefault pragma` also implicitly includes the header file `<som.hh>` if it has not already been included.

- A command-line switch can implicitly insert `#pragma SOMAsDefault(on)` at the beginning of the file. For example, with VAC++ for OS/2

```
#pragma SOMAsDefault(on)

class A {
    int a;
};

#pragma SOMAsDefault(pop)

class B {
    int b;
};
```

Figure 1. SOMAsDefault pragma

```
#include <som.hh>

class A : public SOMObject {
    int a;
};

#pragma SOMAsDefault(on)
class B {
    int b;
};
#pragma SOMAsDefault(off)

class C: private B {
    int c;
};
```

Figure 2. DirectToSOM classes

or Windows, this switch is `/Ga`; for VAC++ for AIX, it is `-qsom`.

- You can use explicit inheritance from `SOMObject` or another DirectToSOM class. For example, all three classes—A, B, and C—are DirectToSOM classes, as shown in Figure 2.

Defining a DirectToSOM class using the `SOMAsDefault` pragma is known as *implicit* or transparent mode, whereas defining a DirectToSOM class by inheriting from another DirectToSOM class (including `SOMObject`) is known as *explicit* mode.

Using DirectToSOM C++ Classes

Once you have defined a DirectToSOM class, what can you do with it? You can create SOM objects statically or dynamically, as simple objects, arrays, or as embedded members of other classes, or anywhere else that the declaration of a C++ object is valid. Figure 3 shows some simple examples (the `SOMDefine` pragma will be discussed shortly).

Most C++ rules and syntax apply to DirectToSOM 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 DirectToSOM C++ class has a SOM release order that by default will contain all member functions and static data members introduced by the class, including those with private and protected access, in the order of declaration. In general, virtual functions that override virtual

```
#include <som.hh>

// SOM class
class OuterSom : public SOMObject {
  public:
    int I;
  private:
    class InnerCPP {
// pointer to SOM object in nested native class
OuterSom *ptr;
    };
};

#pragma SOMAsDefault(on)

// SOM class
class Outer2Som {
  public:
    // embedded SOM member in SOM class
    OuterSom somobj;
  private:
    // embedded SOM class
    class Inner2Som {
int I;
    } somobj2;
};

#pragma SOMAsDefault(off)

// native class
class Outer3Cpp {
  public:
    // embedded SOM member in native class
    Outer2Som somobj;
};

#pragma SOMDefine(OuterSom)
#pragma SOMDefine(Outer2Som)
#pragma SOMDefine(Outer2Som::Inner2Som)

// array of SOM classes
OuterSom array_of_som[100];

int main(void)
{
    // automatic SOM class instance
    Outer2Som outer2_obj;

    Outer3Cpp *ptr = new Outer3Cpp;
    ptr->somobj.somobj.i = 10;

    // dynamic SOM class instance
    OuterSom *ptr2 = new OuterSom;
    ptr2->I = 10;

    // pointer to SOM class member
    int (OuterSom::*pm);
    pm = &OuterSom::I;
    (ptr2->*pm) = 15;
}
```

Figure 3. Using DirectToSOM C++ (samples.cpp)

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 which case you can add new release order elements anywhere in the class, but you must add their names to the end of the list.

C++ instance data members in a DirectToSOM 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 through the SOM Application Programming Interface (API). This scheme allows new data members to be added without requiring recompilation of any code outside the class under the following conditions:

- The declaration order of public and protected data within a class is not changed
- New members are added after any pre-existing members of the same access

All DirectToSOM C++ class function and data members access is performed through the SOM API, rather than the statically defined compiler constructs used by standard C++. This provides for both RRBC and an implementation-independent object model.

```
#include <som.hh>

class Som1 : public SOMObject {
};

class Som2 : public SOMObject {
};

// valid hierarchy: all SOM classes
class Som3: private Som1, protected Som2 {
};

// valid hierarchy: all SOM classes
class Som4: virtual public Som1,
    virtual private Som2 {
};

class nonSom1 {
};

// invalid hierarchy: mixing SOM and native
class mixed : public nonSom1, private Som1 {
};

// invalid hierarchy: Som2 non-virtual
// and appears twice
class Som5 : private Som3, public Som2 {
};

// invalid hierarchy:
// Som2 non-virtual in Som3, so appears twice
class Som6 : private Som3,
      virtual public Som2 {
};

// valid hierarchy:
// Som2 virtual in all bases
class Som7 : protected Som4,
     virtual public Som2 {
};
```

Figure 4. Valid and invalid class hierarchies

Basic Concepts

This section discusses some basic concepts that are important to understand when working with DirectToSOM C++.

Inheritance

SOM does not support an inheritance tree containing anything other than SOM classes. For DirectToSOM C++, this implies that a class hierarchy must contain all SOM or all native C++ classes; it does not support a mixed hierarchy.

SOM also does not permit multiple subobjects of the same type within an inheritance tree. The corresponding DirectToSOM rule is that a class may appear multiple times within a hierarchy only as a virtual base. In other words, only a single occurrence of each non-virtual base class is allowed within a SOM hierarchy. The compiler will issue a warning for each multiple occurrence of a non-virtual base class in a SOM class hierarchy. `SOMObject` is a special case because it is implicitly treated as a virtual base. To illustrate these restrictions, Figure 4 shows various combinations of valid and invalid class hierarchies.

SOM Class Data Structures

For each SOM class implementation, the DirectToSOM C++ compiler must generate several data structures and export three symbols for use by the SOM runtime. The exported symbols are `<class>ClassData`, `<class>CClassData`, and `<class>NewClass`. These symbols are used by

class clients to create and manipulate a SOM class and its instances.

The compiler should only generate these structures and symbol exports once per class implementation, otherwise there would be wasted storage and possible duplicate declaration problems. This is a sensible rule; the problem is determining when to generate the structures. In other words, how does the compiler determine, when parsing a SOM class definition, whether the implementation or the client code is being compiled? For classes that have at least one out-of-line function, the implementation is defined as the file where the definition of the first non-static out-of-line member function is defined. The compiler generates the SOM class data structures and symbol exports as part of compiling this file. This ensures that the class data structures are only defined once for that class implementation.

In Figure 5, the SOM class data structures for class A will be generated with the file that contains the definition for the member function `show`.

However, for classes that have all inline or no member functions, the compiler has no way to determine where to generate the structures. In such cases, you must explicitly indicate where the structures should be generated using the `SOMDefine` pragma. This is why the `SOMDefine` pragmas are used in the earlier example. Without them, the compiler would not generate the SOM class data structures for classes `OuterSom`, `Outer2Som`, and `Outer2Som::Inner2Som`. This would result in link errors due to unresolved implicit references to these structures in the function `main`.

The compiler will generate the SOM class data structure and symbol exports each time it encounters this pragma for a given class. Therefore, it is not a good idea to include the pragma with the class definition, but rather, in a separate file. Otherwise, the compiler will generate the structures each time the header file is parsed, resulting in wasted storage and possible duplicate definition link errors.

Although the above discussion may seem somewhat irrelevant, it is important to understand how DirectToSOM C++ classes are defined. Having duplicate definitions or no definitions for the SOM class data structures is one of the most common, and typically among the first, programming problems encountered when using DirectToSOM C++.

```
#include <som.hh>

class A : public SOMObject {
  private:
    int I;
  public:
    A() { I = 0; }
    void show();
    void set_i(int newvalue) { I = newvalue; }
    int get_i() { return I; }
};
```

Figure 5. SOM class data structure

Linking

As part of creating the SOM class data structures, the DirectToSOM C++ compiler supplies the address of each function and static data member in the class to SOM. This implies that all function and static data members must be defined by link-time because there are external references to them. If you do not supply definitions for all such members, unresolved reference errors will occur at link time. This is different from native C++, where you do not need to define a member unless it is explicitly referenced in the program. If you simply turn SOM mode on for a given class and attempt to create a library, you may discover that some methods are missing implementations that would not have mattered in native C++.

Default Constructor

You should always supply a default constructor for a DirectToSOM class. While you may not use this constructor explicitly in your application, many of the SOM frameworks, such as Distributed SOM (DSOM), require that one be present. In addition, SOM programs written using other languages typically depend upon a default constructor being available. If you are working strictly within DirectToSOM C++ only, and not using any of the frameworks, then technically you do not need to supply it. However, it is best to get in the habit and avoid bugs later on. While the SOM RRBC support makes it easy to add one if needed, runtime errors caused by a missing default constructor can sometimes be difficult to track down.

Header Files

As you have probably noticed, the DirectToSOM C++ header files used so far all have a file extension of `.hh`. Always put DirectToSOM C++ classes in a file with a `.hh` extension. This is not simply a convention; it is required for Interface Definition Language (IDL) generation. Also with regard to header files, you cannot mix the C++ bindings `.xh` header files with the DirectToSOM C++ `.hh` header files in a single compilation unit. The reason is that each provides different definitions of classes such as `SOMObject`. Note that you can mix programs compiled with these different headers at runtime and share SOM objects between them, but you cannot mix them at compile time.

Name Mangling

SOM is case insensitive, so all names presented to it must be unique without respect to case. In particular, class names cannot differ only by case. In order to ensure that unique DirectToSOM C++ names are also unique in SOM, class and member names are subject to a case-insensitive conversion:

- Upper case letters are converted to the lower case equivalent, with a z that precedes the lower case z
- `z_` is used to mean lower case z

Thus `Hello` becomes `zhello` and `ZebraClassZz` becomes `zzebrazclasszzz_`. This converted name is known as the SOM name, as opposed to the C++ name. For example, `zhello` is the default SOM name for the C++ class named `Hello`.

SOMObject Methods

The `SOMObject` base class, from which all DirectToSOM C++ classes derive, defines ten special methods to which certain C++ methods are mapped. It is worth knowing that this mapping is taking place, particularly because the mapped C++ methods are considered overrides of the `SOMObject` methods, rather than newly introduced methods in the class. For a given class `X`, C++ class methods, if supplied, are mapped to the ten special SOM object methods as shown in Figure 6.

Metaclasses

The model that SOM supports is similar to the Smalltalk® model, because classes are not purely syntactic entities as in C++, but are themselves objects. SOM class objects, created at runtime as required by the client, are used for creating and manipulating instances. Class objects support a variety of methods for creating and querying objects, such as determining the size of class instances, whether a method is supported by a given class, and whether a given object is a member of that class.

Figure 7 illustrates this model. As shown at the top of the figure, a native C++ class is a syntactic entity whose definition is compiled into the program object. A native C++ class has no representation except for the source code that defines it. However, with the SOM model, as shown at the bottom of the figure, a SOM class is also an object that exists at runtime. Each SOM class object is an instance of a special class, called a *metaclass*, which by default is the class `SOMClass`. In the same way that a class defines the behavior of its instances, a metaclass defines the behavior of its instances, which are class

```
X()                                somDefaultInit
~X()                               somDestruct
X(X&)                              somDefaultCopyInit
X(X const &)                       somDefaultConstCopyInit
X(X volatile &)                    somDefaultVCopyInit
X(X const volatile &)              somDefaultConstVCopyInit
operator=(X&)                      somDefaultAssign
operator=(X const &)               somDefaultConstAssign
operator=(X volatile &)            somDefaultVAssign
operator=(X const volatile &)      somDefaultConstVAssign
```

Figure 6. Special SOM object methods

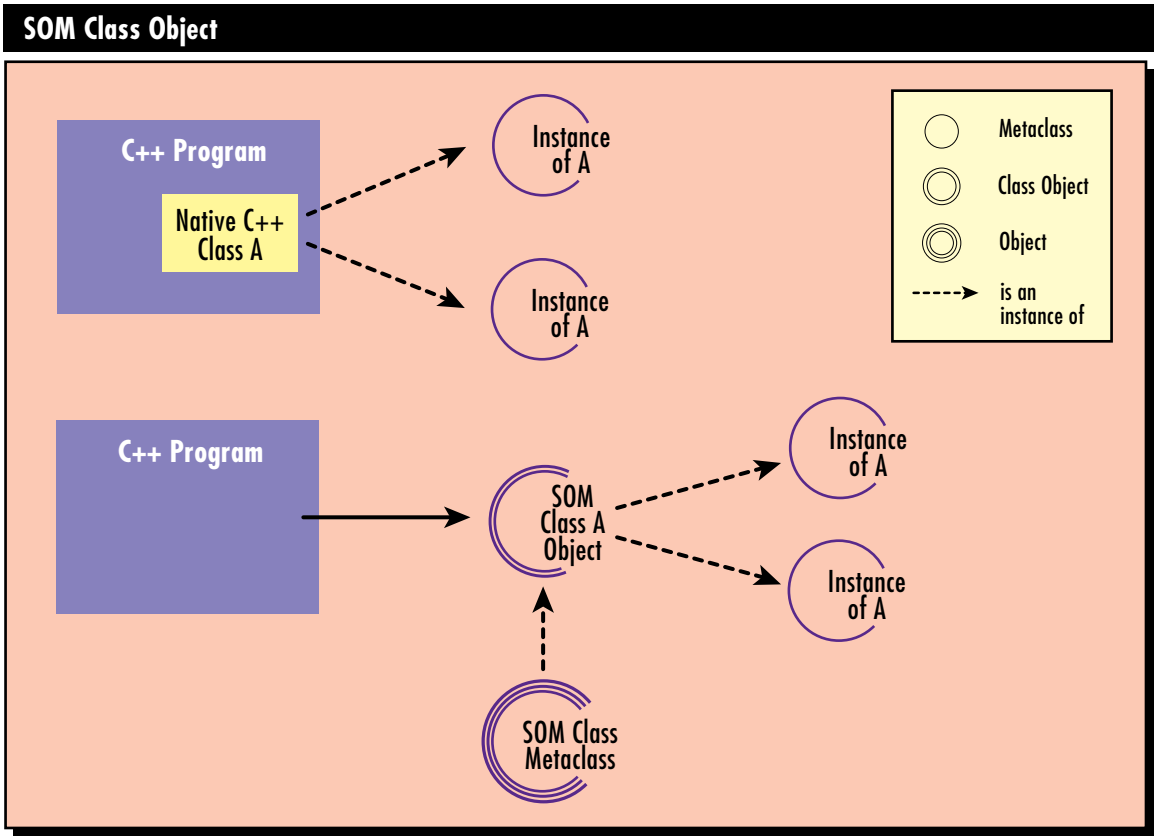

Figure 7. SOM class object

objects. For example, the class A can define the method `foo`, in which case `foo` can be invoked against all instances of A.

Class object methods typically deal with the creation and destruction of class objects. For example, `SOMClass` defines methods such as `somNew`. Since the class object A is an instance of the class `SOMClass`, the method `somNew` may be invoked against that class object in order to create a new instance of A.

If this is the first time you have been exposed to the concept of a metaclass, it may seem a little strange at first compared to native C++. It is not a complicated concept—it is really just a difference in how things are done in the model (although the metaclass concept is much more flexible than the native C++ model). For example, with native C++, you can create instances by invoking the `new` operator, which is applied to a class name. With SOM, the corresponding operation is to invoke the method `somNew` against the appropriate class object. In general, you do not need to deal much with metaclasses when working with SOM, but it helps to understand the concept.

Conclusion

This overview has described how DirectToSOM C++ classes are defined and used, and some of the basic programming considerations for using DirectToSOM C++. For details on these and other SOM-related topics, see *Programming with DirectToSOM C++* published by John Wiley & Sons.

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 non-native object model support for IBM's VisualAge C++ language products. She is the author of three books and numerous articles on programming language-related topics. She has a BSc in Computer Science from the University of Victoria, British Columbia.