# DirectToSom C++ and SOMobjects

Scott Danforth
shd@austin.ibm.com
4/11/96

## Introduction

DirectToSOM C++ is based on the following simple principle: when you define a C++ class that descends from SOMObject, the compiler implements it using SOM. This let's you use all of C++ to define and use SOM classes, which is great news for programmers of the Workplace Shell and other SOM-based frameworks. For example, you can now have SOM objects as local variables on your C++ stack, with full constructor/destructor support.

Two object-oriented programming languages are currently supported by DTS compilers; C++ and IBM OO-Cobol. And, others might be in the future. But, the remainder of this paper focuses on DTS C++ because SOMobjects tools now provide specific support for DTS C++.

These DTS C++ tools are based on SOM IDL, and are referred to as the ***hh*** and the ***hc*** emitters. They support generation of a DTS C++ header file that represents IDL interface and type definitions, and generation of a corresponding DTS C++ implementation template file -- a file containing stubs for the DTS C++ member functions needed to implement SOM classes that support IDL interfaces. To assist code evolution, the *hc* emitter performs incremental update on previously existing implementation template files when you change your IDL.

Users of any particular tool (including a DTS C++ compiler) should expect to find answers to most of their "how do I do this?" questions in the documentation or support forums associated with that specific tool. Therefore, this paper does not provide an in-depth presentation of programming in DTS C++. Most DTS C++ programming questions should be answered by your DTS C++ compiler's documentation. (-:

This paper primarily highlights the *hh* and *hc* emitters. It shows how they support use and implementation of SOM classes whose objects' interfaces are defined using IDL.

## The Big Picture

There are many possible scenarios for using a DTS C++ compiler. In most cases, you will use it to compile source code that you write. It may also (some day) be possible to use the Visual Age C++ VisualBuilder or other tools to produce DTS C++ source code, but this is a topic beyond the scope of this paper. The interesting top-level decisions concern what previously developed class libraries to use and whether or not you are going to create a class library for others to use.

There are basically two kinds of class libraries that you can use and create with DTS C++: native mode C++ class libraries, and SOM class libraries (in general, a mixture of the two is also possible).

To discuss the possibilities, the following terminology is useful. C++ compilers normally use their own, proprietary internal representation for objects, method tables, etc. We call the resulting objects ***native mode*** because they can be used only by code native to the same compiler whose code creates the objects. In contrast, ***DTS mode*** objects are SOM objects programmed using a DTS compiler. These objects can be accessed using the public SOM API from any programming language or compiler that can call externally defined functions. A DTS C++ compiler lets you control which mode is used to implement different C++ structs and classes in your program.

There are two important limitations imposed on DTS mode classes. The first is that a DTS mode class can inherit only one copy of any given ancestor class. This reflects how SOM inheritance works. To strictly satisfy C++ semantics, there should be only one path to each ancestor of a DTS mode C++ class, or, if there are multiple paths to an ancestor (due to multiple inheritance diamond tops), the ancestor should be a virtual base class in all of these paths.

DTS C++ compilers should issue (only) a warning when C++ inheritance semantics is violated by a SOM class hierarchy, and should support a pragma to allow the warning to be turned off in specific cases.

The second limitation is that any given C++ class must either be fully native mode (i.e., none of its ancestors are SOM classes), or fully DTS mode (i.e., all of its ancestors are SOM classes). DTS C++ compilers issue an error if this restriction is violated.

Because of the second limitation, you don't have to explicitly declare whether a subclass is to be compiled using DTS mode. This is determined by the parents of the subclass. Explicit guidance is required only for root classes (that is, classes without parents). The default in this case is native mode, and various command line switches and pragmas are provided for changing this.

Consult your DTS C++ compiler's documentation to learn more about the above details.

## Native Mode Class Libraries

Native mode C++ class libraries are either published in source form, or are published in binary form for use with a specific compiler. Accompanying this are C++ headers that declare the classes (and other types) important for using the class library.

Although DTS C++ compilers provide mechanisms (for example, compile line switches) that allow using DTS mode while compiling source code that contains no DTS pragmas, these mechanisms are intended for code written with knowledge of their use. In most cases, not all of the source for a native C++ class library should be converted to DTS mode. For example, C++ structs without virtual functions may be intended by their designer to be interoperable with C language routines that use C structs. If these C++ structs were compiled in DTS mode, the resulting objects would be SOM objects with an initial method table pointer. As a result, they would not have the structures originally intended by their designer. For reasons such as this, it is best to assume that you will always compile native mode class libraries in native mode. This is the default behavior provided by DTS C++ compilers.

So, native mode class libraries are used in DTS C++ just as they are used in any C++ development environment.

If you subclass from some of the provided classes, you will write a new C++ header for your new subclasses and include the C++ headers for their parents into your new header prior to the declaration of the new subclasses. In general, you'll include your new header into another file that defines the implementations for the member functions that are overridden and introduced by your new subclasses.

If you are going to publish your class library, you package it up (in source or binary form) and additionally provide the C++ headers that describe your classes.

## SOM Class Libraries

In contrast, SOM class libraries are published in binary form and are accompanied by IDL that declares the interfaces and other types important for using the class library.

If some of the non-interface (i.e., non-class) types needed for use of a class library are not completely defined in the IDL, language-specific headers must also supplied. The main reason for discouraging this approach is that instead of providing a single language neutral source for type information, the publisher of the SOM class library must anticipate all the possible languages from which the class library might be used and provide special support for each of these languages. For example, the DSOM class framework doesn't provide self-contained IDL. It originally provided top-level C and native mode C++ headers (**somd.h** and **somd.xh**). Later, to support DTS C++, a new, top-level **somd.hh** include file was provided. It’s best if the IDL describing a SOM class library is self-contained.

But, the DTS C++ compiler doesn't understand IDL (it only understands C++ and a handful of DTS pragmas). So, how do you use a SOM class library from DTS C++ when the classes are described by IDL? The answer is provided by the *hh* emitter. Given IDL as input, the *hh* emitter produces a DTS C++ header file (with an **.hh** extension) that represents all the important information in the original IDL file by using C++ and several special DTS pragmas.

**Note:** The SOMobjects Developer Toolkit provides two different kinds of C++ bindings: DTS C++ bindings (produced by the *hc* and *hh* emitters), and native mode C++ bindings (produced by the *xc*, *xh*, and *xih* emitters). Native mode bindings can be used with any standard C++ compiler, but they don't provide transparent C++ use and definition of SOM classes as enabled by DTS C++. The DTS and native mode bindings are quite different, for reasons that will be explained below.

The SOMobjects Toolkit provides top-level **.hh** include files for SOM and DSOM (these file are named **som.hh** and **somd.hh**). They define primitive SOM types and DSOM types that are not defined in IDL, and include DTS C++ headers for the SOM and DSOM classes (generated from IDL using the *hh* emitter).

Once you've created the DTS C++ headers for the SOM classes you are using (and all their ancestors), you could include the necessary headers into a C++ header that you write for your new subclasses. You would then create an implementation file that includes this new header and defines the necessary DTS C++ member functions.

Alternatively, you could write an IDL interface definition for your new subclass and use the *hh* emitter to produce your DTS C++ header. This alternative approach enables you to define and use new IDL data types in your code. (You cannot currently define IDL data types in DTS C++.)

**Note:** This same **.hh** file (generated from your IDL) serves to support both your implementation code, and code written by users of your class. Special implementation bindings (as used to support C and native mode C++ implementations for SOM classes) are not required by a DTS compiler.

To then create an implementation, you would use the *hc* emitter to create an implementation file, and then fill in the resulting member function stubs.

If your objective is to create a new SOM class library for others to use, you need to provide IDL to represent the functionality of the new SOM classes you have implemented.

If you write your own **.hh** header file, the DTS C++ compiler can process this header and output the corresponding IDL. Consult your DTS C++ compiler's documentation to learn more about this procedure. We don't currently recommend this approach, however. If your users need to see IDL, we recommend that you create this IDL yourself.

If you write the IDL interface for your class according to some simple guidelines (described below) and use the *hh* and *hc* emitter to create an implementation, your original IDL can be published. This is an important practical consideration given limitations in the IDL currently generated by DTS C++ compilers.

It is natural to consider the possibility that some users of your class library may also have DTS C++ compilers. If so, they can use the *hh* emitter to create their own DTS C++ headers from the IDL you provide them.

As another alternative, you might provide users with hand written DTS C++ headers instead of IDL. The main reason for doing this would be that your class library is intended for use only by other DTS C++ users. Normally, this will be the case if your headers use non-IDL data types (for example, C++ arrays, C++ unions, C++ pointers to members, etc.). Note: It is very easy for this to happen when you write your own C++ headers; when you start with IDL, this can't happen..

## Using SOM Classes

The following simple program illustrates use of SOM classes from DTS C++. In this example, the **somEnvironmentNew** function creates an initial SOM environment for the executing process and returns a pointer to the SOM class manager (the class manager is an instance of the SOM class, **SOMClassMgr**, which is a subclass of **SOMObject**). The **somPrintSelf** method (available on all SOM objects) is then invoked on the class manager.

```
// filename: ex1.cpp
#include <som.hh>
main()
{
   SOMClassMgr *cm = somEnvironmentNew();
   cm->somPrintSelf();
}
```

This simple program provides a basic test for the output of the *hh* emitter and the DTS C++ compiler. It causes the DTS C++ compiler to process the emitted **.hh** files **somobj.hh**, **somcls.hh** and **somcm.hh**. The content of **.hh** files is illustrated later.

To compile the above program, your DTS C++ compiler needs to find **som.hh**, which is installed in the SOMobjects Developer Toolkit include directory. Also, the DTS C++ headers generated from **somobj.idl**, **somcls.idl**, and **somcm.idl** are needed. The SOMobjects Developer Toolkit includes a command script named **somhh.cmd** that goes into the Toolkit include directory and generates **.hh** headers for all of the IDL files found there. Among other things, this script executes the command:

```
sc -shh -mnoqualifytypes -musexhpass -S1000000 *.idl
```

This command invokes the SOM compiler, which first parses input IDL to create various data structures, and then passes these to the *hh* emitter (found in **emithh.dll**) which produces the corresponding **.hh** header file.

The reason for the **-mnoqualifytypes** switch is that the *hh* emitter uses nested C++ classes to implement name scoping corresponding to modules in IDL. Normally, the SOM compiler prepends module names to interface names (to create what are called C-scoped names), but the switch prevents this, as required to support the desired mapping approach. Recent versions of the SOM compiler provide special support for the *hh* emitter, and don't require explicit indication of this switch. The **-musexhpass** means that the emitter should emit **C_xh** passthru statements if there are no **C_hh** passthru statements specified in the IDL. The **-S** switch increases the string table size, and is sometimes useful.

The *hh* emitter is a framework emitter that uses the output template file **cpp.efw**. The **SMINCLUDE** environment variable determines which **cpp.efw** file is used. Because DTS C++ compilers sometimes provide these files, you should always check your **LIBPATH** and **SMINCLUDE** environment variables to make sure you are using the latest versions of **emithh.dll** and **cpp.efw**. The *hc* emitter is really a number of different emitters that operate together. The main emitter is named **emitdtm.dll**, and is supported by the **dtm.efw** template file. You can edit **dtm.efw** to make minor changes in the format of emitted **.cpp** files. The other emitter used when **-shc** is indicated on the command line is the **emithc.dll** emitter. This is the emitter that performs incremental update. You can see what emitters are running by using a **-v** switch on the **sc** command line.

In this paper, we generally assume that the latest tool versions are found in the SOMobjects Developer Toolkit, but, in reality, things are more complicated since SOM has many current sources. On OS/2, SOM comes with Warp, with the OS/2 Developers Toolkit, with the Visual Age C++ compiler, and with the SOMobjects Toolkit. And the levels of SOM in these different channels can be different. For example, the version of SOM originally included in Warp was a special build done in Boca, not by the SOM developers in Austin. Typically, SOM developers can only support official versions of SOM provided in the SOMobjects Toolkit. Also, due to an internal IBM reorganization, it now appears that there may no longer be a separate SOMobjects Toolkit product (whose install config process at least tried to avoid replacing later kernel versions with earlier ones). Exactly who provides what and how is likely to be confusing for a while.

Because DTS C++ support is still evolving, it is important for DTS C++ programmers to be using the most recent versions of the *hh* and *hc* emitters. For this reason, they will be made available (in a self-contained package) by their developers to programmers that request them. It is likely that these will provide the best support for DTS C++ programming in the near future. The package will be named dtsmin.zip and will be updated periodically. This plus the latest Visual Age C++ product should provide the best support for DTS C++ in the near future. There's a script named setdts in the package that will place the directory into which you install the package onto the front of all the important paths.

Also, to help you verify what versions of the emitters are being used, version numbers are emitted at the top of generated files. For example here's the top of an **.hh** file.

```
// This header file was generated by the IBM "DirectToSOM" emitter for C++ (V1.125)
// Generated 02/16/96 08:36:34 EST
// The efw file is version 1.65
```

### Using IBM VAC++:

Here is the output from compiling and executing the above main program with the IBM VisualAge C++ compiler, version 3, with the CSD named CTC303 applied.

```
[D:\otp\test\dts\2]icc -yxqnosomvolattr ex1.cpp /B/NOE

IBM VisualAge C ++ for OS/2, Version 3 (C) Copyright IBM Corp. 1991, 1995. - Licensed Materials -
Program Property of IBM - All Rights Reserved. IBM(R) Linker for OS/2(R), Version 01.00.05
(C) Copyright IBM Corporation 1988, 1995.
(C) Copyright Microsoft Corp. 1988, 1989.
 - Licensed Material - Program-Property of IBM - All Rights Reserved.

Object Modules [.obj]: /NOE  /pmtype:vio /base:0x10000 +
Object Modules [.obj]: "ex1.obj"
Run File [ex1.*]: "ex1.exe"
Map File [ex1.map]: ""
Libraries [.lib]:
Definitions File [nul.def]:

[D:\otp\test\dts\2]ex1.exe
{An instance of class SOMClassMgr at address 000C1D90}
```

Before compiling the above example, you should make sure that the **INCLUDE** environment variable used by **icc** reaches the most recent SOM include directories. By default, VAC++ links DTS C++ programs containing SOM classes with the **somtk.lib** library. In the past, some versions of **somtk.lib** contained multiple definitions for some symbols. The VAC++ linker, **ilink**, complains about this, which is the reason for using **/B/NOE** to pass the **/NOE** switch through to **ilink**. You shouldn't need this switch when linking with an up-to-date somk.lib.

**Note:** The reason for the **-yxqnosomvolattr** switch concerns the signatures for the **_get** and **_set** methods associated with IDL attributes. This fixes a bug in the initial versions of VAC++ and should be used until the next version of VAC++ provides this fix by default. It is essential to use this flag if you are using **noset**, **noget**, or **nodata** attribute modifiers in your IDL (explained below).

### Using MetaWare High C/C++:

Here is the output from compiling and executing the above program with the MetaWare HighC/C++ compiler.

```
[D:\otp\test\dts\2]hc ex1.cpp q:\projects\s259549a.deb\lib.os2\som.lib
MetaWare High C/C++ Compiler R2.8a
(c) Copyright 1987-95, MetaWare Incorporated
w "d:/otp/include.os2/somcls.hh",
L104/C24(#692): SOMClass & SOMClass::somAssign(SOMClass &)
|      at "d:/otp/include.os2/somcls.hh",L104/C24
|    overrides SOMObject & SOMObject::somAssign(SOMObject &)
|      at "d:/otp/include.os2/somobj.hh",L77/C25
|    because they have the same SOM external name 'somDefaultAssign';
|    however, they have different argument lists, and this may be a problem.
w "d:/otp/include.os2/somcm.hh",
L59/C27(#692): SOMClassMgr & SOMClassMgr::somAssign(SOMClassMgr &)
|      at "d:/otp/include.os2/somcm.hh",L59/C27
|    overrides SOMObject & SOMObject::somAssign(SOMObject &)
|      at "d:/otp/include.os2/somobj.hh",L77/C25
|    because they have the same SOM external name 'somDefaultAssign';
|    however, they have different argument lists, and this may be a problem.
w (#657):       (info) How referenced files were included:
|    File d:/otp/include.os2/somobj.hh from q:/projects/s259549a.deb/include.os2/som.hh from ex1.cpp.
|    File d:/otp/include.os2/somcls.hh from q:/projects/s259549a.deb/include.os2/som.hh from ex1.cpp.
|    File d:/otp/include.os2/somcm.hh from q:/projects/s259549a.deb/include.os2/som.hh from ex1.cpp.
No errors   2 warnings

Operating System/2 Linear Executable Linker
Version 2.02.001 Jun 09 1994
Copyright (C) IBM Corporation 1988-1993.
Copyright (C) Microsoft Corp. 1988-1993.
 All rights reserved.

Object Modules [.obj]: /noi /e /a:16 /bas:0x10000 /pm:vio +
Object Modules [.obj]: D:\METAWARE\ABI3\lib\startup.obj +
Object Modules [.obj]: ex1.obj +
Object Modules [.obj]: q:\projects\s259549a.deb\lib.os2\som.lib +
Object Modules [.obj]: ,

[D:\otp\test\dts\2]ex1.exe
{An instance of class SOMClassMgr at address 010A1D90}
```

Before compiling the above example, make sure that MetaWare's **hc.cnf** configuration file (in their bin directory) includes the line

```
ARGS=-Hipname=INCLUDE
```

so the directories in the **INCLUDE** environment variable are searched before MetaWare's include directories. Then by setting **INCLUDE** (e.g., by using the setdts command provided by the dtsmin.zip package), you can guarantee that the compiler finds the most recent versions of **som.hh** and the emitted DTS C++ headers for the SOMobjects Developer Toolkit classes. The **som.lib** link library is explicitly included on the command line because HC/C++ doesn't provide a default.

The warnings generated during compilation should go away in future versions of High C/C++. They are a result of differences in the way that MetaWare and VAC++ handle assignment operators for SOM objects. A later example focuses on this specific issue.

## A DSOM Usage Example

Here is an example that shows client DTS C++ code written for a DSOM sample. The methods in the Stack interface take an Environment parameter, but an environment is never explicitly passed to methods in DTS C++. Instead, the DTS C++ compiler passes the current value of __SOMEnv. This is always available within methods that receive an Environment parameter (think of __SOMEnv as being similar to "this", which is available within methods to access the implicit target argument). Other functions (like main, below) can introduce and initialize a local variable of this name. Also, VAC++ (but not HighC/C++) provides a global variable named __SOMEnv. It is not initialized to actually point to an Environment structure, however, but you can initialize it the same way the code below initializes the local variable of this name.

```
#include <somd.hh>
#include "stack.hh"
#include <stdio.h>

boolean OperationOK(Environment *ev);

int main(int argc, char *argv[])
{
    Environment ev;
    Environment* __SOMEnv = &ev;
    Stack *stk=(Stack *) NULL;
    long num = 100;

    SOM_InitEnvironment(__SOMEnv);
    SOMD_Init(__SOMEnv);
    stk = (Stack *) somdCreate(__SOMEnv, "Stack", TRUE);

    /* Verify successful remote stack object creation and use */
    if ( OperationOK(__SOMEnv) ) {
        while ( !stk->full() ) {
            stk->push(num);
            somPrintf("Top: %d\n", stk->top());
            num += 100; }

        /* Test stack overflow exception */
        stk->push(num);
        OperationOK(__SOMEnv);

        while ( !stk->empty() ) {
            somPrintf("Pop: %d\n", stk->pop()); }

        /* Test stack underflow exception */
        somPrintf("Top Underflow: %d\n", stk->top());
        OperationOK(__SOMEnv);
        somPrintf("Pop Underflow: %d\n", stk->pop());
        OperationOK(__SOMEnv);

        stk->push( -10000);
        somPrintf("Top: %d\n", stk->top());
        somPrintf("Pop: %d\n", stk->pop());

        stk->somFree();

        if ( OperationOK(__SOMEnv) ) {
            somPrintf("Stack test successfully completed.\n"); }
    }
    SOMD_Uninit(__SOMEnv);
    SOM_UninitEnvironment(__SOMEnv);

    return(0);
}
```

```
boolean OperationOK(Environment *ev)
{
    char *exID;

    switch (ev->_major) {
        case SYSTEM_EXCEPTION:
            exID = somExceptionId(ev);
            somPrintf("System exception: %s\n", exID);
            somdExceptionFree(ev);
            return (FALSE);

        case USER_EXCEPTION:
            exID = somExceptionId(ev);
            somPrintf("User exception: %s\n", exID);
            somdExceptionFree(ev);
            return (FALSE);

        case NO_EXCEPTION:
            return (TRUE);

        default:
            somPrintf("Invalid exception type in Environment.\n");
            somdExceptionFree(ev);
            return (FALSE);
    }
}
```

# Defining a new SOM class

This example defines a new SOM class named **Ex2**. As mentioned earlier, two different approaches are available for implementing new SOM classes in DTS C++. In this paper, we will only consider the route that begins by writing IDL. It is not necessary to start with IDL, but this is the approach that is familiar to all current SOM programmers, and it offers important benefits for those interested in using IDL to publish CORBA compliant interfaces and support DSOM use.

### Start with an IDL interface definition:

```
// filename ex2.idl
#include <somobj.idl>
interface Ex2 : SOMObject {
  attribute long l;
  implementation {
    dtsdefaults;
    releaseorder: _get_l, _set_l;
  };
};
```

As usual in SOM IDL, the implementation section provides valuable information about how the SOM class that supports the defined interface is implemented. The **dtsdefaults** modifier results in header files that allow the DTS C++ compiler to automatically generate copy constructors, assignment operators, and a destructor for the class. Both the *hh* and *hc* emitters provide special support for **dtsdefaults**. ***<u>As a general guideline, we strongly recommend its use if your're using the hc emitter</u>***. The lack of any explicit modifiers for the attribute **l** results in the DTS compiler automatically generating the necessary **_get** and

**_set** method implementations. The **releaseorder** modifier supports RRBC. The following command generates the necessary header and implementation template files from this IDL.

```
sc -shh;hc ex2
```

The resulting implementation template file, **ex2.cpp**, appears as follows:

```
// Generated from ex2.idl at 04/06/96 09:25:20 EST
// By IBM DTS C++ implementation template emitter version 1.1
// Using hc.efw file version 1.1
#include <ex2.hh>

/* * Begin implementation of interface Ex2 */

// default ctor for dtsdefaults
Ex2::Ex2()
{
}
```

This file can be compiled unchanged to produce a complete implementation for **Ex2**. In general, of course, actual definitions for member functions will be needed. But, to conclude this example, we add the following code to **ex2.cpp**

```
main()
{
  Ex2 e;
  e.somPrintSelf();
}
```

and compile the resulting file using VAC++ (and appropriate environment variable settings) as follows:

```
icc ex2.cpp
```

The executable **ex2.exe** is created, which, when exected, produces the following output:

```
{An instance of class Ex2 at address 00048860}
```

**Note:** The main program above creates a SOM object as a local (stack) variable. This is a very useful capability, because stack allocation is both simpler and more efficient than heap allocation. Native mode C++ bindings cannot support this capability.

### Another example:

Here is another example that uses more of IDL.

```
// filename: ex3.idl
#include <somobj.idl>
interface Ex3 : SOMObject {
    typedef SOMClass::SOMClassSequence classes;
    attribute classes cs;
    void initWithLong(in som3InitCtrl ctrl, in classes ic);
    // an initializer that takes a sequence of classes
     implementation {
        dtsdefaults;
        initWithLong: init;
        somPrintSelf: override; // output the cs sequence
        releaseorder: initWithLong, _get_cs, _set_cs;
    };
};
```

The **init** modifier declares **initWithLong** as a SOM *initializer*. This corresponds to a C++ constructor that will be emitted into the resulting **ex2.cpp** file:

```
// Generated from ex3.idl at 04/06/96 09:59:29 EST
// By IBM DTS C++ implementation template emitter version 1.1
// Using hc.efw file version 1.1
#include <ex3.hh>

/* Begin implementation of interface Ex3 */

// default ctor for dtsdefaults
Ex3::Ex3()
{ }

// an initializer that takes a sequence of classes
Ex3::Ex3(classes* ics)
{ }

// ouput the cs sequence
::SOMObject* Ex3::somPrintSelf()
{ }
```

The above implementation template could be filled out and augmented with a simple test program as follows:

```
/*
 * This file was generated by the SOM Compiler.
 * Generated using:
 * SOM incremental update: 2.24
 */

// Generated from ex3.idl at 04/06/96 09:59:29 EST
// By IBM DTS C++ implementation template emitter version 1.1
// Using hc.efw file version 1.1
#include <ex3.hh>

/* Begin implementation of interface Ex3 */

// default ctor for dtsdefaults
Ex3::Ex3()
{ cs._length = cs._maximum = 0; }

// an initializer that takes a sequence of classes
Ex3::Ex3(classes* ics)
{ int i;
      cs._maximum = cs._length = ics->_length;
      cs._buffer = (SOMClass**) SOMMalloc(cs._length * sizeof(SOMClass*));
      for (i=0; i<cs._length; i++)
              cs._buffer[i] = ics->_buffer[i];
}

// output the cs sequence
::SOMObject* Ex3::somPrintSelf()
{ int i;
      somPrintf("{An Ex3 object with: ");
      if (cs._length == 0) somPrintf(" no classes }\n");
      else { for (i=0; i<cs._length; i++)
              somPrintf("\n\t %s\n", cs._buffer[i]->somGetName());
      somPrintf("}\n"); }
      return this;
}

 main()
{
      Ex3 x1;
      x1.somPrintSelf();
      Ex3::classes ps = (SOMClass::__ClassObject)->somGetParents();
      Ex3 x2(&ps);
      x2.somPrintSelf();
}
```

The output from the above example appears as follows:

```
[D:\SHD\SOM\DTS\1]ex3
{An Ex3 object with:  no classes }
{An Ex3 object with:
      SOMObject
}
```

## Implementing the DSOM Example Stack Class

In this example, we look at a DTS C++ implemenation of the Stack class for which client code was shown in the previous section. We begin by presenting the IDL, and follow this with the DTS C++ implementation code.

```
// filename stack.idl
#include <somobj.idl>
interface Stack: SOMObject {
      const long stackSize = 10;
      exception STACK_OVERFLOW{};
      exception STACK_UNDERFLOW{};
      boolean full();
      boolean empty();
      long top() raises(STACK_UNDERFLOW);
      long pop() raises(STACK_UNDERFLOW);
      void push(in long el) raises(STACK_OVERFLOW);

      implementation {
            dtsdefaults;
            releaseorder: full, empty, top, pop, push; long stackTop;
            long stackValues[stackSize];
            dllname = "stack.dll";
            memory_management = corba;
      };
};
```

```
#include "stack.hh"

Stack::Stack()
{ stackTop = 0; }

boolean Stack::full()
{     /* Return TRUE if stack is full. */
      return (stackTop == Stack_stackSize);
}

 boolean Stack::empty()
{      /* Return TRUE if stack is empty.*/
      return (stackTop == 0);
}

long Stack::top()
{
      if (stackTop > 0) {
            /* Return top element in stack without removing it from the stack.
            return stackValues[stackTop-1]; }
      else {
            somSetException(__SOMEnv, USER_EXCEPTION,
                              ex_Stack_STACK_UNDERFLOW, NULL);
            return (-1L); }
}

long Stack::pop()
{
      if (stackTop > 0) {
            /* Return top element in stack and remove it from the stack. */
```

```
            stackTop--;
            return (stackValues[stackTop]); }
        else {
            somSetException(__SOMEnv, USER_EXCEPTION,
                            ex_Stack_STACK_UNDERFLOW, NULL);
            return (-1L); }
}
void Stack::push(long el)
{
        if (stackTop < Stack_stackSize) {
            /* Add element to top of the stack. */
            stackValues[stackTop] = el; stackTop++; }
        else {
            somSetException(__SOMEnv, USER_EXCEPTION,
                            ex_Stack_STACK_OVERFLOW, NULL); }
}
```

### An Emitted DTS C++ Header File:

So, far, we've deliberately avoided discussing the content of emitted **.hh** files. One reason was to stress the underlying simplicity of the basic approach, which doesn't really require reference to DTS C++ headers. All you have to do is fill out the member function stubs provided by the implementation template file.

But, of course, it is important to be familiar with emitted **.hh** files. For one thing, when there are problems that result in compilation errors, the **.hh** files often provide the information necessary to resolve problems. Also, if you decide to write your own **.hh** file, you may want to use some of the special DTS C++ pragmas that are used to control various aspects of DTS mode classes.

For these reasons, we now display and discuss the **.hh** file corresponding to **ex3.idl**. We've simplified the file somewhat to remove redundancies, and have numbered the lines for ease of reference.

```
#ifndef _DTS_HH_INCLUDED_ex3
#define _DTS_HH_INCLUDED_ex3

/* Start Interface Ex3 */
// Generated from ex3.idl at 04/06/96 11:18:57 EST
// By the IBM DTS C++ header emitter version 1.142
// Using cpp.efw file version 1.74

#include <som.hh>

class Ex3 : public ::SOMObject {
#pragma SOMClassName(*, "Ex3")
#pragma SOMNoMangling(*)
#pragma SOMCallstyle (idl)

public :
#pragma SOMAsDefault(off)
typedef ::SOMClass::SOMClassSequence classes;
#pragma SOMAsDefault(pop)

// dtsdefault ctor
Ex3();
```

```
23
24  // new method: initWithLong
25 typedef void SOMLINK somTP_initWithLong(Ex3 *somSelf, Environment *ev,
26                                      ::som3InitCtrl* ctrl, classes* ics);
27 typedef somTP_initWithLong* somTD_initWithLong;
28
29 // an initializer that takes a sequence of classes
30 Ex3(classes* ics);
31 #pragma SOMMethodName( Ex3(classes* ics), "initWithLong")
32
33  // output the cs attribute
34 virtual ::SOMObject* somPrintSelf();
35
36  classes cs;
37 #pragma SOMAttribute(cs, virtualaccessors)
38
39 #pragma SOMReleaseOrder ( \
40                 "initWithLong", \
41                 "_set_cs", \
42                 "_get_cs")
43
44 #pragma SOMAsDefault(off)
45 typedef struct {
46     SOMClass *classObject;
47     somMToken initWithLong;
48     somMToken _set_cs;
49     somMToken _get_cs;
50 } __ClassDataStruct;
51 #pragma SOMAsDefault(pop)
52 #define SOM_MTOKEN_Ex3(mName) \
53     (((::Ex3::__ClassDataStruct*)&Ex3ClassData)->mName)
54 #define SOM_PARENT_Ex3(obj,mName,icls,mToken) ((icls::somTD_ ## mName) \
55     somParentResolve((somMethodTabs)(Ex3CClassData.parentMtab),mToken))
56 };
57 /* End Ex3 */ #endif /* _DTS_HH_INCLUDED_ex3 */
```

We mentioned earlier that the native mode C++ bindings are quite different from the DTS C++ bindings. One way to understand this is as follows. In both cases, it is necessary to tell the compiler things that it doesn't otherwise know. In the case of native mode C++ bindings, the C++ compiler must be guided in how to invoke a method on a SOM object. So the native mode C++ bindings define member functions that expand to SOM method invocations. In contrast, the DTS C++ compiler knows how to make SOM method calls. But, it doesn't know the SOM names for methods declared in C++ (these are generally different than the C++ names, because C++ method names are mangled to include signature information in support of C++ overloading). Working through the above header file should clarify this.

Lines 1 and 2 are used to guard the contents of the file against multiple inclusions within a single compilation unit.

Lines 4 through 7 provide indicate the levels of the *hh* emitter and the supporting **cpp.efw** file.

Line 9 #includes **som.hh**. This is a specially constructed top-level header that #includes other headers that define SOM's basic types and APIs, and then includes the generated **.hh** files for the three classes provided by the SOM kernel.

Line 11 begins the declaration of the DTS mode C++ class named **Ex3**. The pragma on line 12 is what causes the class to have this name, since, otherwise, that actual name of the class would be **zex3**. This is because, by default, DTS mode class names are mangled to remove upper case letters. The reason is that case cannot be used in IDL to distinguish different identifiers, and it is desired that legal C++ class hierarchies should be mapped to legal IDL interface hierarchies (when possible) without requiring intervention on the part of the C++ programmer. The result, however, is that C++ programmers that write their own header files must take special care if they want their class and method names to appear sensibly to non-DTS C++ users.

Line 13 indicates that method names in the current class should not be mangled, and line 14 indicates that all introduced methods will take an implicit environment parameter. In both IDL and DTS C++, the environment parameter is somewhat like the target object reference because it is not explicitly declared in method prototypes, nor is it explicitly passed as an argument by the DTS C++ programmer. Within methods that receive an environment, it can be accessed using the name **__SOMEnv**. Refer to the DSOM Stack client and implementation examples provided earlier to see code that accesses and manipulates the environment.

Line 16 begins public declarations. The **SOMAsDefault(off)** pragma on line 17 is used for normal type declarations, so C++ structs won't be mapped to SOM classes. This is not structly necessary for the typedef on line 18, since it just introduces a local name for a previously defined type. But, a number of IDL types are mapped to C/C++ structs, so it is important that the *hh* emitter provides this support in general. Line 19 returns the **SOMAsDefault** mode to whatever it was before line 17. (Which is normally off). When **SOMAsDefault** is on, C++ structs and classes without parents will be compiled using DTS mode.

Lines 21 and 22 result from the dtsdefaults modifier placed in the ID, and guarantee that the class will have a default no-argument constructor even if other constructors are explicitly defined (as is the case in this example).

Lines 24 through 27 provide a typedef for the **initWithLong** method. The *hh* emitter provides typedefs for all new methods introduced by a class. This is not needed for normal method invocations, but SOM enables more dynamic capabilities than those provided by C++ (or even DTS C++), and these typedefs are often useful in this regard. For example, the currently available DTS C++ compilers don't support SOM parent method calls. To support these, emitted **.hh** files provide special macros (discussed below) which use these tyepdefs. In particular, the **somTD** typedef form is useful to cast a function pointer as necessary to inform the compiler of the expected arguments of a method implementation. Notice how both the target object reference and the environment pointer appear explicitly in these resulting typedefs.

In this case, the **initWithLong** method is a SOM initializer. These are reflected in emitted **.hh** files as C++ constructors with appropriate signatures, as one line 30. The SOM approach is somewhat more general than provided by C++, since different initializers with the same signature are allowed, overrides are allowed, and objects can be initialized multiple times. Currently, no warnings are provided when two different initializers have the same signature. Future versions of the *hh* and *hc* emitters will hopefully address this issue. But, in the mean time, when starting with IDL, ***a good guideline is to avoid defining multiple initializers with the same signature***.

The pragma on line 31 is used to inform the DTS compiler of the SOM name of the initializer method. Constructors and assignment operators are different from normal C++ methods, so this constructor is not covered by the **SOMNoMangling** pragma on line 13.

Line 34 is the override for somPrintSelf, a method introduced by **SOMObject** (and therefore described in the DTS C++ header file **somobj.hh**).

Line 36 and 37 declare the cs attribute. Attributes are handled in a very special way by DTS C++. By default, public attributes declare nonvirtual public methods (the **_get** and **_set** accessor methods) supported by private data. Data access syntax in DTS C++ then uses the accessor methods instead of direct data access. There are a number of modifiers that can be used in SOMAttribute pragmas to change these defaults. See the documentation for your compiler for complete information, but the following should be supported:

**noset**
: Compiler won't define **_set**; the programmer does it

**noget**
: Compiler won't define **_get**; the programmer does it

**nodata**
: An instance variable will not be introduced to support the attribute accessors

**readonly**
: no **_set** operation will be provided

**virtualaccessors**
: accessor methods will be virtual (they can be overridden)

Note that the **virtualaccessors** modifier is used in the SOMAttribute pragma on line 37. The reason is that this is the default in IDL (where the attribute was defined). If any of the modifiers **noset**, **noget**, or **nodata** is used, the programmer must define accessor methods. When this is done, the **-yxqnosomvolattr** (icc compiler switch) is important, because otherwise the DTS compiler will not accept programmer definitions of these methods. Specifically, the problem is that the VAC++ compiler will expect different overloadings than those provided by the implementation template emitter, which are based on an agreed-to DTS C++ specification.

Lines 39 through 42 provide the **releaseorder**, which supports RRBC.

The remaining lines are provided until the DTS C++ compiler provides better support for parent method calls. The structure definition represents the **ClassData** structure for **Ex3**, which is where SOM stores method tokens. The **SOM_MTOKEN_Ex3** macro is used to access these method tokens, and the **SOM_PARENT_Ex3** macro is used to perform parent method calls. Papers on Workplace Shell programming using DTS C++ explain the importance of supporting parent method calls. The macro defined here is restricted to use in purely single inheritance hierarchies (such as the WPS), and is supported by currently available SOM kernels. SOM kernels supporting a new parent method call model appropriate in general multiple inheritance hierarchies will be made available later this year. A file named **dtsmin3.zip** will be used to package up DTS C++ emitters that use the new parent call with the new SOM kernel that supports it.

### Copy Constructors

Copy constructors for SOM classes are implemented by overriding one or more of the **SOMObject** methods provided for this purpose. These methods are named **somDefaultCopyInit**, **somDefaultConstCopyInit**, **somDefaultVCopyInit**, and **somDefaultConstVCopyInit**, which represent all four C++ overloadings of a class's copy constructors with respect to **const** (or not) and **volatile** (or not). Nobody really knows what the volatile forms are for in the case of SOM classes, but they are provided in **SOMObject** because otherwise subclasses wouldn't be able to define them. Who knows? Maybe you'll have some use for the volatile overloaded forms. But, we suggest that (only) **somDefaultConstCopyInit** be defined by all SOM classes. Assuming that the parents of a class do this, the **dtsdefaults** modifier will allow the DTS C++ compiler to do this without programmer involvement. Although the compiler can provide a reasonable default version, you may want to define your own implementation in some cases. To do this, you simply override **somDefaultConstCopyInit** in your IDL (while continuing to use **dtsdefaults**). The resulting implementation template will include a const copy constructor that you can define.

The reason that only the **const** form needs to be defined is that the SOM kernel and the DTS C++ compiler arrange that if only a **const** copy constructor is defined by a class, then it will be used for both **const** and **non-const** arguments. Here is an example IDL file and a corresponding emitted **.cpp** file that illustrates an explicit definition of **somDefaultConstCopyInit**.

```
#include <somcls.idl>
interface X : SOMObject {
    attribute SOMClass::SOMClassSequence cs;
    implementation {
        dtsdefaults;
        somDefaultConstCopyInit: override;
    };
};

// Generated from t.idl at 04/08/96 08:04:38 EDT
// By IBM DTS C++ implementation template emitter version 1.1
// Using hc.efw file version 1.1
#include <t.hh>

/* Begin implementation of interface X */

// default ctor for dtsdefaults
X::X()
{ cs._length = cs._maximum = 0; }


X::X(const X& arg)
{
    int i;
    cs._length = arg.cs._length; cs._maximum = arg.cs._maximum;
    cs._buffer = (SOMClass**) SOMMalloc(cs._maximum * sizeof(SOMClass*));
    for (i=0; i<cs._length; i++)
        cs._buffer[i] = arg.cs._buffer[i];
}
```

## Assignment Operators

Assignment operators for SOM classes are similar to copy constructors, in that there are four special methods introduced by **SOMObject** for this purpose: **somDefaultAssign**, **somDefaultConstAssign**, **somDefaultVAssign**, and **somDefaultConstVAssign**. As with copy constructors, only the **const** form, **somDefaultConstAssign**, needs to be defined. And, again, the **dtsdefaults** modifier allows the DTS C++ compiler to provide an appropriate default. If you need to define this operator, you simply override **somDefaultConstAssign** in your IDL (while continuing to use **dtsdefaults**).

However, your implementation template file will not include a user-defined **operator=** when you override **somDefaultConstAssign**. This is because it was decided that DTS C++ should implement the semantics of C++ with respect to a user-defined **operator=**, and this is not a good solution for assignment in SOM class hierarchies. So, DTS C++ compilers provide an alternative syntax whose meaning can be unambiguously be interpreted as requesting an override for one of **SOMObject**'s assignment methods. Currently, there are two different syntaxes for this: one supported by MetaWare's High C/C++, and the other supported by VAC++. For a number of reasons, the MetaWare form is more desirable, and VAC++ will support this in the future. But, currently, an #ifdef is used in the emitted template file to distinguish the two different forms. So, when you override **somDefaultConstAssign**, there will be a corresponding #ifdef in your implementation template file, and you must define the branch appropriate to the compiler that you are using.

Here is an example IDL file and a corresponding emitted **.cpp** file that illustrates explicit definition of **somDefaultConstAssign**. In this example, we fill in both branches of the #ifdef, so you can see the difference between the HighC/C++ approach (which uses `__EXTENDED__SOM__ASSIGNMENTS__`) and the VAC++ approach (which doesn't).

```
#include <somcls.idl>
interface X : SOMObject {
    attribute SOMClass::SOMClassSequence cs;
    implementation {
        dtsdefaults;
        somDefaultConstAssign: override;
    };
};


// Generated from t1.idl at 04/08/96 08:15:03 EDT
// By IBM DTS C++ implementation template emitter version 1.1
// Using hc.efw file version 1.1
#include <t1.hh>

/* Begin implementation of interface X */

// default ctor for dtsdefaults
X::X()
{ cs._length = cs._maximum = 0; }

#ifdef __EXTENDED__SOM__ASSIGNMENTS__
X::X& somAssign(const X& fromObj)
{
    int i;
    cs._length = fromObj.cs._length; cs._maximum = fromObj.cs._maximum;
    cs._buffer = (SOMClass**) SOMMalloc(cs._maximum * sizeof(SOMClass*));
    for (i=0; i<cs._length; i++)
        cs._buffer[i] = fromObj.cs._buffer[i];
    return *this;
```

```
}
#else
SOMObject* X::somDefaultConstAssign(som3AssignCtrl * ctrl, const SOMObject* fromObj)
{
    X &fromX = (X&)(*((X*)((void*)fromObj)));
    int i; cs._length = fromX.cs._length; cs._maximum = fromX.cs._maximum;
    cs._buffer = (SOMClass**) SOMMalloc(cs._maximum * sizeof(SOMClass*));
    for (i=0; i<cs._length; i++)
        cs._buffer[i] = fromX.cs._buffer[i];
    return (SOMObject*)((void*)this);
}
#endif
```

As can be seen from this example, the HighC/C++ form makes use of a pseudo operator named **somAssign** that is given the right types for an **X** assignment operator. The result is that DTS C++ programmers don't need to use type casting to implement assignment operators. The HighC/C++ compiler currently issues warning messages about the argument and return types used for somAssign (refer to the first example, where the MetaWare compiler output was shown), but **MetaWare** will be removing these in future versions of the compiler.

In constrast, the VAC++ form uses the actual methods introduced by **SOMObject**. Because C++ typing rules don't allow changing the types of these methods in subclasses as required to reflect their actual purpose as assignment operators, the programmer needs to perform casts -- first, to access X's data in the fromObj, and then to return the assigned object as the result.

# Conclusion

Both from the standpoint of programmer ease and execution efficiency, DTS compilers can provide the best support for creating and using SOM classes. The SOMobjects Developer Toolkit supports DTS C++ by providing top-level DTS C++ header files and emitters to generate DTS C++ headers and implementation templates for SOM classes described using IDL. When DTS C++ programmers use IDL to describe the interfaces for their classes, and use the dtsdefaults modifier, these emitters make it simple to implement SOM classes using DTS C++.

# Current Problems

Perhaps the most serious remaining problem with the DTS C++ compilers concerns the lack of correct support for IDL modules by DTS C++ compilers. Although the .hh file provides a temporary workaround for this, it only supports using instances of such classes -- i.e., the workaround is not sufficient to support DTS C++ implementation of classes or subclassing from classes whose corresponding interface is defined within an IDL module. This is a known defect, and will be fixed in the near future. The most usual symptom of this defect is that parent classes will not be located during class construction.

Also, the hh emitter has some limitations. It doesn't support anonymous sequences used as method arguments in IDL. And, it doesn't support nested modules (i.e., IDL modules contained within other modules). To get around the anonymous sequence problem, just use a typedef to give a name to the desired sequence. Hopefully, support for nested modules is not a serious issue, since it is not likely to be fixed in the near future.