CenturaPro™

Visit us at www.ProPublishing.com! Hot Ideas for Centura® Developers

The Secret's Out: Centura *is* Dynamic

Gianluca Pivato

Dynamic objects instantiation with object pointers is one of the most overdue improvements we could ever ask for in SQLWindows. It has been years now that developers have been asking for this feature—since the days of SQLWindows 4. Now it's possible—and it actually always has been!

SQLWindows 16/32

For years developers have had to find creative ways to overcome a particular limitation in SQLWindows' implementation of object-oriented programming. We use window handles as object pointers, but it takes too many resources—plus Windows can't handle too many windows. We use "callback" functions to pass an object up to a base class as a parameter, but the code becomes less flexible. We also use global predefined arrays of objects with the index in the array as a pointer, but it has other obvious limitations and problems. I've always thought that if I only had dynamic object instantiation and pointers, my OOP code would be great!

Wouldn't it be nice to be able to write something like this?

```
Object Handle: hObject
CMyClass: Test

!Create the object
Set hObject = SalUdvCreate( CMyClass )
!Using late bound
Call hObject.CMyClass.fMethod()
Set hObject.CMyClass.m_sStringMember = "Ciao"
! Using early bound
Set Test = hObject.CMyClass
Call Test.fMethod()
Set Test.m_sStringMember = "Ciao"
!Free the object
Call SalUdvFree( hObject )
```

Too good to be true, right? In fact, it can't be done. But we can get pretty close:

```
Window Handle: hObject
CMyClass: Test

!Create the object
Set hObject = XSalUdvCreate( CMyClass )
! Using early bound (late bound isn't possible)
Call XsalUdvLock( Test, hObject )
Call Test.fMethod()
Set Test.m_sStringMember = "Ciao"
Call XsalUdvRelease( Test, hObject )
!Free the object
Call XSalUdvFree( hObject )
```

Actually, you can also do much more than this. You can create an object using the class name as a string; and

Continues on page 3

June 1999

Volume 4, Number 6

PRO Publishing

Centura *is* Dynamic . . .

Continued from page 1

you can test an object handle to see if it's directly created from a class or if the class appears anywhere in the inheritance tree. You can reference an existing object (object pointers) and even use the keyword "this" (or "self" for Delphi programmers) to get a pointer to the current object. How many times have you needed to pass the current object as a parameter? Now it's easy!

How does it work?

I've often pondered in my spare time about what it would take to add dynamic object instantiation and object pointers to SQLWindows. After all, I thought, the basic code to create an object from a class definition at runtime is already there. It happens each time a locally declared object has to be allocated and de-allocated and when an array of objects gets truncated or extended. It also happens when you create a window from a visual class derived from a functional class.

Using "SalArrayCreateUdv, SalArrayDestroy" in CDLLI11.DLL (or CDLLI15.DLL for CTD 1.5), you can actually create an array from a UDV class at runtime:

```
hArray = SalArrayCreateUdv( CClassName )
```

The declaration in the external functions is as follows:

```
Function: SalArrayCreateUdv
  Description:
  Export Ordinal: 0
  Returns
    Window Handle: HWND
  Parameters
    Template: TEMPLATE
```

So, now I can create an empty array from a UDV class at runtime. I can extend it and truncate it using SalArraySetUpperBound( )—all SalArray* functions accept a window handle as the array parameter—but I can't really use it unless I pass it to an external DLL . Even doing so, I would only be able to access the object's attributes. It's not much use for dynamic instantiation.

The solution is to use templates, which is similar to C++ casting. I used a check-out/check-in mechanism to switch the dynamic object's attributes with a "known" object in the outline acting as a template or casting variable. This is why I need the XsalUdvLock( )/ XsalUdvUnlock( ) pairs.

When you use XSalUdvLock( ), you "check out" the dynamic object into the template object, which is recognized by the SQLWindows engine, since it already existed at compile time. See the similarity with C++:

C++	XSAL
CMyClass* T;	Window Handle: H CMyClass: T
T = new CMyClass;	Set H = XSalUdvCreate(CMyClass) Call XSalUdvLock(T, H)
T->Method();	Call T.Method()
T->Property = 0;	Set T.Property = 0
Function(T);	Call Function(T)
delete T;	Call XSalUdvUnlock(T, H) Call XSalUdvFree(H)

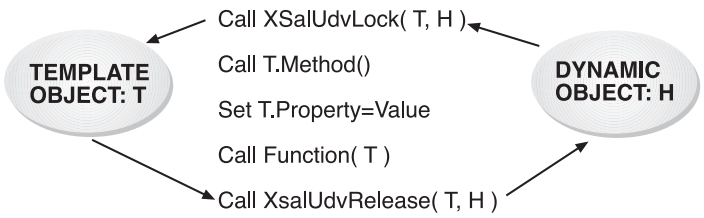

Figure 1. Lock/Unlock pairs.

The "template method" allows me to use an object not present in the outline at compile time during runtime. But what happens in XSalUdvCreate? The creation of the dynamic object can actually be accomplished in three ways, and I tried all three of them before finding the right one.

Creating dynamic objects

My first approach was to use SalArrayCreateUdv( ) to create a UDV array with only one item and use the array handle as the object handle. In the Lock/Unlock calls I could use the SWinArrayUdvAddress( ) call to get the memory address of the object to move its memory block to and from the template object.

This works, but it has some limits. After a certain number of arrays CTD crashes (I went up to 30,000), and there's no way to distinguish between an object pointer and a dynamic object.

My second approach was to use the array as a Class Factory (COM style). I created an internal list of dynamically created arrays for each class type, with only one item. One class, one array, and no more. Whenever I needed to create a new object, I allocated a block of memory for the object (the size is retrievable using SalOutlineClassSize( )), moved the new item memory block into the newly allocated memory block. To delete a dynamic object, I moved the object back into the array, truncated the item to free all the associate memory (strings and other arrays), and then re-extended the array of one item to be ready for the next new object request.

This approach allowed me to use my own structure for the object memory allocation in order to store extra information, such as the class handle (HITEM) and the type of handle. This way I could use the same handle for dynamic objects and object pointers. The object pointer is only some kind of "proxy" to an existing UDV. But this approach had a major flaw!

When SQLWindows creates a new object from a class, it initializes all its members to their default initial value. This is very important for numbers and dates. String handles are set to null and array members are created. And here's the problem. The new item in the dynamically created array (the class factory) always had the same array handles, which means that different instances created from the same class were sharing the same array members! It's a very bad case of data corruption.

I thought of creating the new item only when requested by XSalUdvCreate( ), resetting all its values and destroying it right after so that each new object would receive unique array handles. But there would be a problem in the destruction of the objects. To destroy the dynamic object, I need to move it back into the dynamically-created array. But the array doesn't have any item anymore, and if I extended the array by one again, a new set of embedded array handles would be created, and I would have memory leaks.

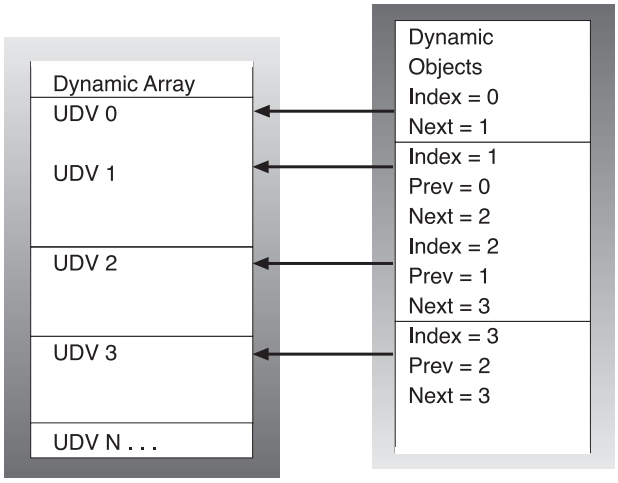

Figure 2. The memory structure.

The final solution

The third and final approach, which worked, is actually a mix of the two previous approaches.

When XSalUdvCreate( ) is called, I look up the class HITEM in a table of existing dynamically created arrays. If the class was never used before, I create a new array using SalArrayCreateUdv( ) and add it to the memory table. The second step is to extend the array by one to create the object. The third step is to allocate a small memory block in which I store the handle type, the index in the array, the memory address of the UDV for object pointers, and other useful information. The last part consists of adding this memory block to a "daisy chain" (or linked list) of objects. The object pointer—what you declare as "Window Handle" in the outline—is the memory address of this "proxy" structure.

The fun comes with the destruction of the object. In memory we have two structures: the dynamically created array, which holds each instance of our dynamic objects; and the linked list of dynamically created objects. In **Figure 2** you can see a graphic representation.

When a dynamic object needs to be freed, I have to eliminate the corresponding item in the array. The only way to do that is to truncate the last item in the array using SalArraySetUpperBound( ); but the object that has to be released might not always be the last one created. My solution was to swap the last item in the array with the item to delete and then truncate the array.

To swap two items in a UDV array, I need only get their memory address using SWinArrayUdvAddress( ) and physically move the memory block using memmove( ). Then I have to update the object that was previously allocated to the last item in the array. Looking at Figure 2, if I need to delete UDV0, and I swap UDV3 (the last one), the dynamic object with Index = 3 would become unusable. The best approach is to also swap the two dynamic objects in the linked list; being a linked list, it's easy and fast.

The result is that the order in the linked list always reflects the order in the dynamically-created array.

Creating object pointers

An object pointer is the same as a dynamic object without the corresponding item in the dynamically created array. It's like an "empty shell" or "proxy." Object pointers are kept in a separate linked list to avoid interfering with the tight correspondence between the dynamic objects list and the array.

I have two functions for object pointers:

```
Window Handle: hObject
Set hObject = XSalUdvGetObject( OBJ )
Set hObject = XSalUdvGetObjectThis()
```

The first one creates an object pointer out of an existing object; the second creates the pointer out of the currently executing object.

The object pointer contains a pointer to the memory address of the UDV it's referring to. The memory address of a UDV is in the first four bytes (LONG) of the HUDV structure. To retrieve the HUDV of the currently executing UDV, I simply use SalUdvGetCurrentHandle( ).

Object pointers need to be freed using XSalUdvFree(

hObject). In case you forget, XSal2 will free everything you have allocated during runtime when the application stops. Plus, there a couple of diagnostic functions that allow you detect the number of currently allocated objects.

Using dynamic objects and object pointers

Now you know how the underlying memory management works. But the usage is a different story.

As I mentioned earlier, the only solution I could find was to use the "template method." This requires you to "check-out" the dynamic object into a known template and to check the object back in when you're done. An object can be checked out only once. After it has been checked out, you should only use the template to which the dynamic object was assigned.

The "check-out" procedure is composed of two steps: First, save the memory block of the template object into a temporary buffer associated with the dynamic object, and then copy the dynamic object into the template buffer. This way the runtime engine is working as usual with the known object; but in actuality it's working on a different set of values. The "check-in" procedure does exactly the opposite, ensuring that whatever has changed in the template goes back into the dynamic object, and that the template memory block gets restored to its original values. This way, when runtime frees the template (for example, exiting from a local code block), it doesn't destroy the content of the dynamic object. The persistence is assured.

The same technique works for object pointers; instead of copying the memory block out of the dynamically-created array, I copy it straight from the object. See what is possible now that was impossible before:

```
Functional Class: CParent
  Instance Variables
    String: m_sName
    ChildObject: CChild
  Functions
    Function: Test
      Actions
        Call ChildObject.SetParentTest(
          XSalUdvGetObjectThis( CParent ), "Luca" )
        Call SalMessageBox( m_sName, "", 0 )
Functional Class: CChild
  Functions
    Function: SetParentTest
      Parameters
        Window Handle: p_hObjectParent
        String: p_sValue
      Local variables
        Parent: CParent
      Actions
        Call XSalUdvLock( Parent, p_hObjectParent )
        Set Parent.m_sName = p_sValue
        Call XSalUdvRelease( Parent, p_hObjectParent )
        Call XsalUdvFree( p_hObjectParent )
```

A child object is able to know about its parent object.

More useful functions

You can use other very useful functions to achieve much better OOP than has ever been possible with SQLWindows:

```
XSalUdvCreateFromString( "ClassName" )
XSalUdvIsObjectOfType( hObject, ClassName )
XSalUdvIsDerivedFrom( hObject, ClassName )
```

Using XSalUdvCreateFromString( ), you can pass the class name as a string parameter, which means you can actually decide the class to use at runtime. The object handle datatype is always a window handle anyway.

XSalUdvIsObjectOfType( ) allows you to detect the class that was used to create the object. This function is of great help when you need to test the handle before casting it into a template (check-out):

```
Set hObject = XsalUdvCreateFromString( sClassName )
If XSalUdvIsObjectOfType( hObject, CClass1 )
  Call XSalUdvLock( T1, hObject )
  Call T1.Method1()
  Call XsalUdvRelease( T1, hObject )
Else If XSalUdvIsObjectOfType( hObject, CClass2 )
  Call XSalUdvLock( T2, hObject )
  Call T1.Method2()
  Call XsalUdvRelease( T2, hObject )
```

XSalUdvIsDerivedFrom( ) returns TRUE if the class is anywhere in the dynamic object's inheritance tree. This is useful if you need to pass the object as a parameter to a function that has a parameter declaration of a base class.

A sample application

The sample code I've included is simple. It was intended to show the reliability and speed of dynamic objects.

The sample application creates a variable number of objects and initializes them with a string and number. Using the arrow buttons you can browse the collection of objects. You can change the value of each individual object by just typing in the datafields.

This sample application could have been written using dynamic arrays, but looking at my source code, you'll notice that the object collection isn't declared anywhere; it's created dynamically!

Can every dynamic object be a COM automation object?

One of the potential, relatively easy enhancements of the technique I've illustrated is to be able to attach any kind of code to the dynamic object structure, the one that holds

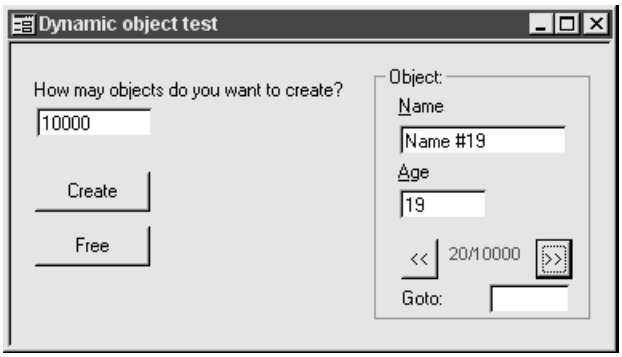

Figure 3. The sample application.

the index to the dynamically created array or the pointer to the existing object.

It's possible to implement the IDispatch interface and make the object immediately become an automation object. The IDispatch has to do two basic things: return an ID from a member name and dispatch requests for the member.

GetIdsOfNames

Since we can't read the definition of the class at runtime, we build one at runtime. Each time a client requests a name, we look it up in a table; if it exists, we return the ID; if it doesn't, we simply add it.

Invoke

This is a little more difficult. This function is used to read and write the properties and to execute the methods exposed from the automation object. To read and write the properties, we can simply use SalOutlineVarOffset( ) to retrieve the offset of the instance variable from the memory address of the UDV.

To execute a function call is a little more difficult. The technique is to build the call using the parameters list and "fool" SalCompileAndEvaluate( ) by manually changing the context string. Remember that we're talking about dynamically-created objects, which means that we don't have the object name in the outline at compile time. So we can't simply use "Call OBJ.Method(p1, p2)". We only have the function name, not the object name.

The context string (the last parameter in SalCompileAndEvaluate( )) holds, among many things, the memory address of the currently executing UDV and the class HITEM. We have both! The solution is to retrieve the context string and to change those two LONG numbers in there. After that SalCompileAndEvaluate( ) will think that it's running in the object scope.

What to do with the technique

The first use that comes to mind is the Microsoft Script Control. It's a COM class that runs on COM clients, implements VBScript and Jscript, and is freeware. You can use VBScript/Jscript to script your CTD applications, but you must register the objects that the script recognizes. One way to register them is to use the newly released XSalCOM, but that means that *all* your objects become published COM objects—they go into the Windows registry.

Using dynamically created objects, you can simply pass the object IDispatch* pointer (at this point it's the same as the object handle) to the Script controls and you're done. I'll write a more detailed example of this very useful feature for a future issue of *Centura Pro*.

How good is it?

I know that this new OOP enhancement, dynamic objects, is not the same as C++, Visual Basic, or Delphi. And I know that with access to the SQLWindows/32 engine source code, it would be possible to do much better than this. However, this is a pretty good implementation that enables a lot of previously impossible things—important things! Many of the sophisticated OOP features of a great 3GL like Java or C++ are now available in the world's best 4GL, SQLWindows. They always were "available"—they were just waiting for someone to expose them. **CP**

Download DynObj.ZIP from this issue's Table of Contents at www.ProPublishing.com or find it on this month's Companion Disk.

Gianluca Pivato designs LANs, WANs, databases, software, and anything that interacts with the system. Developer of XSal and XSal2, he recently developed XSalCOM, a C++ application that turns Centura applications into Automation Servers. His company is Pivato Consulting, inc., and he can be reached at gianluca@pivato.com.