Sharing Code and Objects Between Delphi and C++

Conrad Herrmann, Staff Engineer, Borland International

Delphi's unique blend of component-oriented development methodologies and lightning-fast compilation have made it extremely popular since its introduction. Delphi draws Windows programmers from many different environments -- Visual Basic, Clipper, Fox Pro, dBASE, Paradox, and of course C++. As programmers begin using Delphi, it's natural that they would want to carry their existing code with them. C++ programmers are fortunate that Object Pascal shares with C++ a common object interface model and function calling convention. This makes it especially easy to share code and components between the two languages.

The purpose of this material is to introduce the Delphi programmer to ways in which C++ programs and components can be integrated into Delphi programs and components, and vice-versa. The course covers:

- ways to share functions and objects between C++ and Delphi.
- how to combine C++ and Delphi code into one program using object files or dynamic libraries.
- how to add Delphi forms to OWL and MFC applications

Sharing Functions

Calling a C++ function from Object Pascal, or vice-versa, is entirely straightforward, since both languages provide the same basic facilities. The following two lines declare the same function prototype:

```
C++:     extern "C" int __stdcall MyFunc();

Pascal:  function MyFunc: integer; stdcall;
```

To call the function, you use the following syntax:

```
C++:     int x;
         x = MyFunc();

Pascal:  var
           x: Integer;
         begin
           x := MyFunc;
         end;
```

Make sure you match the calling conventions used in the function declarations. It also helps to declare the calling convention you're using explicitly, since you never really know if a user might change the default calling conventions in the IDE.

Unless you have some reason to choose otherwise, use the stdcall calling convention because it is the "standard" system calling convention. One exception is when you want to have variable numbers of parameters, in which case you must use the cdecl calling convention.

In C++ code, declaring the function as extern "C" will prevent the C++ compiler from giving the function a "mangled" symbol. This symbol will be what the Pascal compiler expects when it sees a cdecl function.

If you intend to pass parameters to your function, you need to choose types that are available in both languages. Pascal's Comp, Currency, Extended, Strings, Sets, method closures, and open arrays do not map directly to C++ data types. You can pass a String by value as a PChar parameter as long as the receiving function does not attempt to release the memory held by the string.

The reference modifier in C++ formal parameter lists corresponds to the VAR keyword in an Object Pascal procedure declaration. Both indicate that the variable is to be passed by reference. For example:

```
C++:    int MyFunc( int &x );

Pascal: function MyFunc( var x: Integer ) : Integer;
```

Packaging

Putting Library Routines in a DLL

The simplest way to make your code available to be used by *any* other language is to put the C++ code in a DLL, and export the public functions from the DLL. If your code represents some kind of an engine, or a logical middleware layer, this is usually the best approach since it allows two projects to interact with each other without being too intimately aware of each others' inner details.

To export a C++ function from a DLL, just add the export attribute to its definition. For example,

```
 int stdcall export  MyFunc();
```

To import a function from a DLL into Object Pascal, declare it with the _external_ specification:

```
function InitObject : TMyObject; stdcall; external 'MYLIB.DLL';
```

Linking OBJs

If you want to link the C++ object file directly into an Object Pascal program file, you can do so by using the $L directive. This directive tells the Delphi compiler to include the .OBJ file into the final .EXE or .DLL. For example,

```
function MyFunc : Integer; stdcall;
{$L myfunc.obj}
```

When to package a component as a DLL and when to use OBJs

It makes most sense to package the code as a DLL when:

- the C++ code is a complete, whole subsystem.
- there is a lot of code in the C++ component.
- the C++ code makes use of the C runtime library (RTL).

It makes sense to link the code together using .OBJs when:

- there isn't much C++ code at all.
- you don't need to worry about the RTL interactions.

Sharing Objects between Borland C++ and Delphi

When you're using two different object-oriented languages, you can define two levels at which the languages accommodate each other's objects: *sharing objects* and *sharing classes*.

To *share objects*, code in one language should be able to

- keep a pointer to an object that was created in another language, and
- call methods of the object specified by the pointer.

To *share classes*, code in one language must be able to

- create an instance of a class specified in another language,
- destroy an instance of a class and remove its memory from the heap, and
- derive new classes from the class.

Delphi and Borland C++ can share objects because you can pass compatible object pointers between the two languages, and you can call methods on an object implemented in one language from the other. In the following section I'll present an example of how you can manipulate a C++ object from Pascal, and then present an example of how you can manipulate a Pascal object from C++.

Accessing and Manipulating a Pascal object from C++

How can a C++ program call a method on a Delphi object? The first thing to do is make sure the two languages have an understanding of what methods the object supports. An object's *interface* is a specification of what methods can be called on an object. If the two languages can agree on an interface that is to be used to talk to the object, then one of the languages can implement the object and either language can call the object's methods.

Just as you describe a function interface by publishing its prototype but not its implementation, we can describe an object's interface by describing (in both Pascal and C++) a class that has no actual implementation--all the methods are virtual and abstract (in C++, we call this "pure virtual").

To define an interface for a Pascal object:

- In Pascal, declare an interface class that contains the methods on the object that you want to share. The methods should be declared abstract and virtual.
- Translate the interface to C++. Use the virtual keyword and the =0 clause to make each method "pure virtual".
- Make sure both languages are using the same calling conventions. Use stdcall by default.
- Derive the object's class from the interface class.

For example, here is a Pascal object whose methods we want to call from C++.

```
TMyObject = class
  procedure DoThis( n: Integer );
  function DoThat: PChar;
  procedure WalkUp;
  procedure WalkDown;
end;
```

Suppose you are interested in calling the DoThis and DoThat methods from C++, but not the WalkUp and WalkDown methods because they are private to the object's implementation. You need to build an interface that defines DoThis and DoThat, and expose that interface by deriving TMyObject from the interface. By convention, interfaces start with the letter *I*, to distinguish them from classes that have real implementations behind them.

```
IMyObject = class
  procedure DoThis( n: Integer ); virtual; abstract; stdcall;
  function DoThat : PChar; virtual; abstract; stdcall;
end;

TMyObject = class(IMyObject)
  procedure DoThis( n: Integer );
  function DoThat: PChar;
  procedure WalkUp;
  procedure WalkDown;
end;
```

Now we can translate the interface to C++:

```
class IMyObject {
  void DoThis( int n) = 0;
  char *DoThat() = 0;
};
```

The Pascal IMyObject is binary-compatible with the C++ IMyObject, because both languages implement virtual method tables the same way. Since I can cast a TMyObject to an IMyObject in Pascal, and I can access IMyObject from C++, then once my C++ code acquires a pointer to an object's IMyObject interface, it can call methods on the object.

For example, the following C++ code calls the DoThis method on the pointer that was returned by the function CreateMyObject.

```
extern "C" IMyObject stdcall *CreateMyObject();

void ExampleFunc()
{
  IMyObject *theObj = CreateMyObject();
  theObj->DoThis(5);
  char *s = theObj->DoThat();
}
```

Before the C++ code can try to call a method on the object it first has to get a pointer to the object. Since the object is implemented in Pascal, this means you need to either

- ask a Pascal function to return it, or
- ask another Pascal object to return it.

Factory Functions

In the example above, the function CreateMyObject is just a plain function, exported from the Delphi DLL (or OBJ) in the manner I discussed previously. It returns an interface of class IMyObject, which it is responsible for creating. (For the purposes of our example, assume that the factory code is in a DLL called ObjUtils and that it needs to be exported.)

```
library ObjUtils;
  exports CreateMyObject;
  function CreateMyObject : IMyObject; stdcall;
  begin
    Result := TMyObject.Create;
  end;
end.
```

Notice that even though the C++ program is asking for the object to be created, it is actually being created (i.e., the memory is being allocated and the class hierarchy is being used) on the Pascal side. This is a simple convention that allows one language to ask another language to create an object. A function like CreateMyObject is called a *factory function*, because its purpose is to manufacture and return objects.

An advantage to using a factory function is that the code that calls the factory does not have to be made aware of *how* the object is made in order to create it, it only needs to know that there exists a factory function that it can ask to create one.

Of course it is possible that you don't want to create a new instance of an object but that you want to get access to one that exists already. To do this, you might end up asking another object for it, or you might

just export a global function to retrieve the object interface. Which you choose depends on what kind of object you're asking for and how many of them there are.

Freeing Delphi objects from C++

You now can create objects, but how do you destroy a Delphi object from C++? Since we used a function to create the object, we could very easily use another function to destroy it.

A different way to do this is to have one of the methods on the interface be a method that destroys the object. For example, you can add a Free method to the object, and add that method to the interface:

```
IMyObject = class
  procedure DoThis( n: Integer ); virtual; abstract; stdcall;
  function DoThat : PChar; virtual; abstract; stdcall;
  procedure Free; virtual; abstract; stdcall;
end;

TMyObject = class(IMyObject)
  procedure DoThis( n: Integer );
  function DoThat: PChar;
  procedure WalkUp;
  procedure WalkDown;
  procedure Free;
end;
```

And in C++:

```
class IMyObject {
  void stdcall DoThis( int n) = 0;
  char stdcall *DoThat() = 0;
  void stdcall Free() = 0;
};
```

The implementation of the Free method simply destroys the object:

```
procedure TMyObject.Free;
begin
  self.Destroy;
end;
```

Using a C++ Object from Delphi

In the last section I discussed how you can use a Delphi object from C++. Now let's consider the case where a Delphi program wants to use a C++ object. I expect that this case is a bit more interesting since programmers using both languages are more likely to be using Delphi for programming a user interface and just want to connect the UI code to their C++ engine code.

It turns out that you can use the same techniques presented in the last section for handling this case, too. The interfaces are the same, but the implementation of the object is now on the C++ side. For example, to implement the same class we described in Pascal above, use a class definition like this:

```
class TMyObject: public IMyObject {
  void DoThis( int n );
  char *DoThat();
  void WalkUp();
  void WalkDown();
  void Free();
}
```

Your C++ program would also need to include a factory function, like the following. Here, I'm assuming the C++ code is in a DLL, so the factory function is exported.

```
IMyObject stdcall export *CreateMyObject() {
  return (IMyObject *)new TMyObject;
}
```

Your Pascal code can import and then call the CreateMyObject function to get a pointer to the object, and it can call any of the methods presented by the interface. When it is finished, your Pascal program can free the object by calling the Free method through the interface.

Subclassing is not for Components

Remember I said there were three things that you need to do to be able to share classes? I'm going to skip that one because we're talking about components.

The thing that makes components what they are (and component-oriented programming so simple) is that objects are self-contained and don't reveal their innards to other objects. But to subclass from a class, you have to know all about its internal implementation. To delve into the guts of an object just to subclass it across language boundaries rather defeats the purpose of using the objects as components.

A Side Excursion Into the Component Object Model (COM)

If all this talk about interface classes sounds fashionable, consider one additional convention: the Component Object Model.

COM is a specification for how objects can surface interfaces and how other objects can interact with them. COM is also the low-level object model that OLE is built on. COM is just one application of the interfacing technique I've been describing, but there are enough books and articles written about it that it's worth mentioning specificly.

All COM interfaces have at least these three methods:

```
virtual HRESULT QueryInterface( riid: REFIID, void **p)=0;
virtual long AddRef()=0;
virtual long Release()=0;
```

All COM objects are reference-counted objects, which means they don't have a Free method. Instead, the object knows how many other objects are pointing to it; when that number drops to zero, the object destroys itself. The AddRef and Release methods increment and decrement the use count of the object.

The rule for using AddRef is that you should call it when you make a copy of the interface pointer to hand off to another object or function, and it is up to the recipient of that copy to call Release on the interface when it is no longer needed.

The QueryInterface method permits an object to provide more than one interface. If you have one interface pointer on the object, you can always ask for another by specifying which interface you want. QueryInterface provides the means for asking for that interface.

These three methods make up the interface class IUnknown from which all other COM interfaces are derived. (It's called IUnknown because if you have an IUnknown pointer you have no idea what kind of object it is, but you know you have one).

COM is the glue that makes OLE work. It is a very big subject and beyond the scope of this talk. If you're interested in learning more about COM, I heartily recommend Kraig Brockschmidt's book *Inside OLE*. Its first few chapters are an excellent primer on the COM architecture. Mr. Brockschmidt also has written several articles for Microsoft Systems Journal.

Using Delphi Forms From C++

If you have a large body of engine code written in C++ and intend to write a new Windows user interface in Delphi, the techniques I've already described will serve you very well. If, on the other hand, you already have a significant amount of user interface code written in C++, it would be a shame to have to throw it out just because you have a new UI builder tool. You don't have to throw out your C++ code. This section of the course covers how you can use Delphi Forms with existing C++, OWL-based, or MFC-based applications.

First, let's understand the scope of the problem. Most business programs have two kinds of user interface surfaces: windows and dialogs. The distinction is largely that windows stay up all the time and generally have more complex graphical elements, and a dialog is modal and usually just has entry fields and other controls.

In the following section, I'll cover both how to use Delphi modal forms as dialogs in a C++ program and how to use Delphi modeless forms as new windows in a C++ program.

Using Delphi Modal Forms to Implement Dialogs

Again, the easiest way to make Delphi code available to a C++ program is to make a DLL and export the important functions. In the case where we want to make a Delphi form that acts like a modal dialog box, this function would be one that opens the form, waits for the user to commit or cancel the form, and returns data to the C++ caller.

You'll want to start by creating a new DLL project and add a form. Pick the "Dialog" form template from the project expert to have the form designer place the default OK, Cancel and Help buttons on the form. Then design the form as you would normally do.

When you are done, go into the form's unit file and add a function that invokes the form as a modal dialog. The function looks like this:

```
function RunMyForm : Integer; stdcall;
begin
  Application.CreateForm(TMyForm, MyForm)
  MyForm.Visible := True;
  Result := MyForm.RunModal;
end;
```

To call the dialog from C++, just call the RunMyForm function in the DLL:

```
extern "C" int stdcall RunMyForm();
main() {
  int x = RunMyForm();
  printf("The result is %d\n", x);
}
```

Make sure you export the function in your Pascal library's **exports** directive, and link your C++ program with the import library.

Using A Delphi Form as a Modeless Window in a C++ MFC or OWL Program

Delphi Forms and MFC or OWL windows are all Windows objects, represented by a window handle. Since an OWL or MFC window can contain any other window so long as it has a window handle, it is really a very simple matter to insert a Delphi Form as the child of an OWL window--just create the Delphi form, take its window handle, and set its parent to be the OWL window. There are a few other issues to deal with, as I'll outline below.

Making an Embeddable Delphi Form

To make the Delphi Form in a DLL, you need to export a factory function that takes the parent window handle as a parameter.

You will also need to change the superclass of your form class from TForm to TEmbeddableForm. TEmbeddableForm can be found on the Conference CD, in the unit EmbForm. The reason for using the TEmbeddableForm subclass is because even though any Windows window can be parented to any other, the current VCL implementation does not expose such a method. TEmbeddableForm implements this method.

To recap the procedure:

- Design your Delphi form.
- Change the derivation of the form so it inherits from TEmbeddableForm instead of TForm. Add EmbForms to the **uses** clause of your program file.
- Write a Factory function in the library that looks like the function below. Export the function using the **exports** statement in your library module.

```
function CreateTForm1( hWndParent: HWND; var pObj: IUnknown ) : HWND; stdcall;
var
  theForm: TForm1;
begin
  Application.CreateForm( TForm1, theForm );
  theForm.ParentHandle := hWndParent;
  Result := theForm.Handle;
  pObj := nil;   // optionally, return another interface to the window, too
end;
```

You'll notice that the factory function returns both a window handle and a pointer to an IUnknown, which in my example is always nil. This second return value is simply a convenient way of allowing the containing window to get both the window handle and some other interface on the window.

Writing OWL C++ Source

If you want to make an OWL-based window that will host a Delphi form, all you have to do is create an OWL TWindow object to wrap the Delphi form's window handle.

In the source code I've included on the Conference CD, I implement just such a class in dlctl.h. TDelphiFormWrapper is an OWL class derived from TWindow that takes the factory function in place of a window class for creating a window. When you make a TDelphiFormWrapper object a child of any OWL window and then create the parent, TDelphiFormWrapper will take care of calling the factory function with the parent's handle when the parent window is created.

The Delphi form wrapper class also takes care of passing on the PreProcessMessage method when it is called, which allows the Delphi form to correctly process tabs, arrows and other dialog navigation keys.

To include the Delphi form in your C++ OWL program, follow these steps:

1. Add the following include to the top of your C++ source file

```
#include "dlctl.h"
```

Dlctl.h defines TDelphiFormWrapper, which is a TWindow-derived window class that uses your factory function to create the window.

2. Define the factory function as an external function in your C++ code, so you can call the factory.

```
extern HWND CreateTForm1(HWND Parent);
```

3. Where you would normally create a child window, instead make an instance of the class TDelphiFormWrapper, and pass the Factory function (CreateTForm1) as the first constructor parameter.

```
TWindow * myWindow =
  new TDelphiFormWrapper(CreateTForm1, 0, 0, 0);
TFrameWindow *MainWindow =
  new TFrameWindow(0, "Sample OWL Window containing a Delphi control",
    new TDelphiFormWrapper(CreateTForm1, 0, 0, 0));
```

4. Link your project with the Delphi DLL's import library, otherwise you'll get an unresolved symbol.

Modifying the OWL C++ Header to Allows Attach/Detach

In order to implement the Delphi RecreateWnd method correctly, the Windows implementation of TWindow should really support public Attach and Detach methods. This is because when you change certain properties in a Delphi form the window handle of the Delphi Form is actually destroyed and recreated.

Version 5.01 supports Attach and Detach, but version 5.0 does not. Therefore, I've included a special version of WINDOW.H from OWL which defines this function. Because the Attach and Detach methods are implemented as inlines, you can use this version of the header file with the version 5.0 OWL DLLs or libraries, since it doesn't change any of the code that exists internal to OWL.

Writing MFC C++ Source

Making an MFC window that will host a form is also straightforward. You just need to be able to create a CWnd object as a child of your main window, and then Attach the window handle you get back from the Factory function.

The code below illustrates how this works. Note that the LoadFrame method actually causes CMainWindow's window handle to be created, so we don't need to call its Create method to ensure the main window is actually created.

```
CMainWindow::CMainWindow() {
  LoadFrame(IDR_MAINFRAME, WS_OVERLAPPEDWINDOW);
  LPUNKNOWN pObj;
  CWnd *child = new CWnd();
  child->Attach( CreateTForm1( m_hWnd, pObj ) );
}
```

Conclusion

This course has been about understanding the similarities between two different languages. As it turns out, since C++ and Pascal are fairly compatible languages, functions and objects can be easily shared between them.

Both languages can share function or procedure symbols, and can link into or with DLLs. This allows you to easily share the code you've already written in one language with the other.

The key concept that makes object sharing possible is sharing interfaces. When an object defines a sharable interface, it allows one language to call methods on an object implemented in another language. Factory functions and Free methods allow you to create and destroy an object from another language. The COM model extends this even further by making it possible for an object to be shared with many different languages simultaneously.

Also, because OWL, MFC and Delphi windows are all deep down just "Windows windows", there really is no reason why you can't plug one into the other. There are a few issues that arise (such as special message processing that one or the other introduces), but I've provided a suite of classes to implement solutions to these issues. There are a number of examples on the conference CD-ROM for you to learn from.

Additional References

This course introduces some new classes to facilitate the interaction between Delphi and C++ class hierarchies (MFC or OWL). The source for these classes is available on the conference CD-ROM, along with some sample programs.

A more in-depth discussion of linking options, object sharing, and issues relating to parameter types can be found in the article, "Using Borland's Delphi and C++ Together" by Alain Tadros and Eric Uber, at href=http://www.borland.com/News/techlib/brick.htmlMACROBUTTON. Reference materials for object file linking, DLL construction, calling conventions and type compatibilities can be found in the Object Pascal Language Guide.