Application Development with C++Builder and Delphi

Written by Charlie Calvert

Overview

C++Builder and Delphi have a unique relationship that allows programmers to easily share code between the two environments. Almost anything you write in Delphi can easily be used in C++Builder, and much of the code you write in C++Builder can be used in Delphi. As a result, programmers can work in either C++ or Delphi, depending on their skill level or inclination.

Despite their similarity, Delphi and C++ each have a unique set of capabilities:

- Delphi's ease of use allows a wide range of programmers to quickly produce high performance, robust components, objects or applications.
- C++Builder, in the other hand, allows the best programmers on your staff to write code in a language known for its power and wide acceptance.

Because you can share code between the two environments, you can allow your staff to use the tools that best suit their current needs. Then later on, you can swap the code back and forth between the two environments.

This paper is divided into two main sections. The first part contains an executive summary suitable for the general reader. The second, and longer, section of the paper contains a more technical analysis of the methods for sharing code between C++Builder and Delphi.

Benefits of Sharing Code Between C++Builder and Delphi

This section lists some of the key benefits gained when you share code between Delphi and C++Builder. These items are listed in no particular order.

- Programmers can work in either C++ or Delphi. You don't have to worry that code written with one tool will not be accessible from inside the other tool. Instead, you can concentrate on finding the fastest way to create usable modules, and then link them in to either C++Builder or Delphi, depending on the needs of your engineers.

- Sharing code between Delphi and C++Builder also promotes code reuse. You can write an object once for a Delphi project, and then reuse it, unchanged, in a C++ project. There is no performance or size penalty associated with writing code in Delphi rather than C++.

- Delphi is extremely easy to use. It offers programmers a simple, easy to understand syntax. Companies can allow programmers to work in Delphi without fear that their code cannot also be used in C++ projects. You can therefore have some of your team working full time in C++, and the rest of the team working full time with the much easier to use Delphi language. The efforts of both programmers can then be combined into a single, high performance executable.

    - Because Delphi is so easy to use, it is often the language of choice during projects that have tight deadlines. Use Delphi to create high performance programs as easily as possible. Any modules or objects created on these types of projects can then later be linked into C++ projects. This means that there is no penalty associated with using Delphi inside your company, even if you are primarily a C++ shop.

Technical Highlights

This section of the paper contains an overview of the key points you need to know about sharing code between C++ Builder and Delphi. It consists of two subsections called:

1. Overview: Using Delphi Code in C++Builder
2. Overview: Using C++ Code in Delphi

When you are through reading these sections you will have a general feeling for how, and to what degree, you can share code between Delphi and C++Builder. In subsequent sections I will review most of these points in more detail.

Before beginning, you should note that Delphi uses Object Pascal, which is widely taught in many schools. C++Builder uses C++, which is probably the most widely used language in corporate America. Most programmers will know one, if not both, of these languages.

Overview: Using Delphi Code in C++Builder

C++ Builder does not care whether the units you add to your project are written in C++ or in Delphi. It will compile either type of unit as if it were native to C++ Builder.

As a general rule, any Delphi 2.01 unit you create that compiles in Delphi compiles unchanged in C++Builder. There are a very few Delphi syntactical elements that will not be accepted by C++Builder. You will generally find that all your Delphi 2.01 or Delphi 2.0 code will compile unchanged in C++Builder. This is an unprecedented level of seamless integration between Delphi and C++.

In particular, the following types of code will compile in C++Builder:

1) Delphi forms.
2) Delphi units containing objects.
3) Delphi units containing procedures, functions, constants, structures, arrays, etc.
4) Delphi components.

You usually do not have to change your Delphi units at all to link them in to C++Builder. There is no need to change the code, to create header files, or to massage your code in any way. You can mix the types of code listed above in any way you like. For instance, you can link into C++Builder one Delphi unit that contains:

- A form
- An object
- Some functions
- A component

The unit can contain any combination of the above types, or of any other valid Pascal types. These types may appear in any combination. For instance, a Pascal unit can contain only an object, or it could contain, both an object and a form, etc. You can link in as many Pascal units as you like, each containing any combination of the above objects. There is no limit on the number of Delphi units you can add to a C++Builder project.

To link Delphi code into C++Builder you simply choose Add to Project from the menus, then browse for files that end in PAS. Once you have added one or more Delphi units to your project, they will compile and link seamlessly with the rest of your C++ code. Use Visual Form Inheritance to modify Delphi forms within C++Builder. You can resynchronize when updates are made to the original Delphi form.

Besides the simple techniques outlined above, advanced programmers can also link Delphi code into C++ by using one of the following techniques:

1) Delphi dynamic link libraries (DLLs), can be easily added to your C++ projects.
2) You can use COM and OLE to add Delphi 2.0 or Delphi 3.0 objects to C++Builder. In particular, you can add Delphi objects to C++ Builder via:
    3) OLE Automation
    4) The basic rules of COM
    5) Creating Delphi ActiveX controls that you use in C++Builder.
    6) Delphi and C++Builder share the same Virtual Method Tables (VMTs). This means you can create an object in Delphi, and use it in C++ Builder via a technique outlined below in the section called "Mapping Virtual Method Tables"

Overview: Using C++ Code in Delphi

C++ supports some syntactical elements such as multiple inheritance and function overloading that make it difficult to directly link all C++Builder code into Delphi. However, you can still share code between the two environments via:

1) COM and OLE. Delphi fully supports both OLE Automation and dual interfaces.
2) DLLs
    3) Direct linking of C++Builder units containing functions.

The most convenient technique listed above is the first, especially if your C++ COM objects support type libraries. Delphi 97 has full support for type libraries, and will be able to generate Delphi units directly from a type library. This means that you can link C++ COM objects that support type libraries directly into Delphi without having to write any extra code. All the necessary files for linking in C++ COM objects that support type libraries will be generated automatically by Delphi.

DLLs are a powerful means of sharing code between Delphi and C++ Builder. Virtually any C++ Builder function that you create can be placed in a DLL and called directly from Delphi. Delphi supports all the common calling conventions such as CDECL, PASCAL, and STDCALL. You can access an object written in C++ from inside Delphi simply by reversing the technique described below in the section entitled "Mapping Virtual Method Tables".

It is possible to link C++ OBJ files directly into Delphi. This is a powerful technique that can be very useful in certain circumstances. However, their are limitations on what you can do with this technology, and it should not be viewed as a primary means of porting code between the two environments.

Summary of the Executive Overview

Before closing this section of the paper, it might be helpful to review some of the key points mentioned so far. This will complete the executive summary for non-programmers who do not wish to delve into the technical details involved in linking code between the two environments.

C++Builder has virtually no limitations on it's ability to use Delphi code. You can simply link your Pascal units directly into C++Builder without modifying them in any way. They will then compile into the same small, tight, high-performance machine code you would expect if you wrote them in C++.

If you use both C++Builder and Delphi in a single project, you can promote code reuse while simultaneously promoting the skills of all your programmers. C++ engineers can work in C++ where they can take advantage of that language's power. Where tight deadlines, ease of use, and a short learning curve drive development, programmers can work in Delphi. With this division of labor, all the programmers on your team will be immediately productive and can reuse each others work.

Your best engineers can work in C++, where they can take advantage of that language's flexibility. Inexperienced programmers, and programmers who are rushed, can work in Delphi. With this division of labor, all the programmers on your team will be immediately productive.

The key point is that the technology outlined in this paper allows your programmers to work in either C++ or Delphi, without fear that the code will not be available for use in all your C++ projects.

A More Technical Analysis

You have now completed the overview of the techniques for sharing code between C++Builder and Delphi. The remaining sections in this paper explore these subjects in more depth. In other words, if you are a manager, you can now put this paper down and go on to something else. If you are a programmer, this is where the paper starts to get interesting. The subject matter in the following sections starts out with the simplest techniques, and becomes increasingly complex as the paper nears its conclusion.

Ground Rules for Linking Delphi Code into C++ Builder

You have already learned that you can link Delphi code directly into C++Builder projects. In this section I will explain how this is possible, and put to rest any fears you might have about engaging in constructing this kind of project.

First, I will lay down three key ground rules.

1) You cannot mix C++ and Delphi code in the same unit. You have to create separate units (modules) for each type of code.
2) You must have the source to the unit you want to use. You cannot link a Pascal binary file (DCU) into a C++Builder project.
3) You cannot edit a Pascal unit while it is open in C++Builder.

Furthermore, you should note that a Pascal unit cannot usually be the project file for a C++Builder application. However, if this one, small module is written in C++, and everything else in the project is written in Object Pascal, then the project is still, by definition, a C++Builder project, even though the code is ninety-eight percent written in Object Pascal.

Here is a detailed description of the simple steps need to link a Delphi unit into a C++Builder project:

1) First choose the **Project | Add to Project** menu option.
2) When the **Add to Project** dialog appears pull down the **Files of Type** combo box at the bottom of the dialog. You will then be able to select files of type CPP, PAS, C, RES, OBJ or LIB. You should select PAS.
3) Browse for the Pascal source file you want and add it to your project. After you have selected the file you want to use, you can click on OK to add it to your project and proceed exactly as you would if this were a C++ file.

The header file for your unit will be generated automatically. You don't have to do anything to the unit to use it in a C++Builder project.

Linking a Delphi Component into a C++Builder Project

If you want to use a Delphi component in a C++ project, you would normally want to first add it to the Component Palette. You do this via the same basic technique you would use in Delphi. In particular:

1) Choose **Component | Install** from the menu.
2) Select the **Add** button from the Install Components Dialog
3) Use the **Add Module** dialog to enter or browse for the name of the PAS file you want to use, that is, for the PAS file that contains the registration procedure for your Delphi component.
4) After selecting the module, click on the OK button in the Install Components dialog. C++Builder will then recompile CMPLIB32.DLL, thereby adding your component to the Component Palette.

After you have integrated a Delphi component into C++Builder, you can then treat it exactly as if it were a native C++Builder component. That is, you can drop it onto a form and manipulate its methods with the object inspector. If you need to reference any of the components methods in your C++ code, then you should use C++ syntax. The details of this syntax can be surmised by viewing the header file created when you added the component to the Component palette.

Some Theory on Linking Object Pascal and C++Builder Code

Delphi is an Object Pascal based development environment, while C++Builder is obviously a C++ based development environment. How is it possible to link Pascal directly into a C++ project?

A good way to begin your exploration of this issue is to understand that Delphi, and Borland C++ 5.0 use the same 32-bit compiler. The difference is simply in how the language is parsed. Delphi parses Object Pascal, tokenizes it, and then passes it on to the compiler. Borland C++ 5.0 parses C++, tokenizes it, and then passes it on to the compiler. Underneath the hood, Delphi 2.0, C++ Builder and Borland C++ are very similar tools. The difference is in how the syntax of the language looks before it is parsed and tokenized.

Delphi finishes its compile and link cycles faster than Borland C++ simply because Pascal is easier to parse, and because Delphi supports a simpler, and more flexible type of binary file (DCUs are easier to make than OBJs.)

Since C++Builder uses the same compiler as Delphi, the road is obviously at least half way open to finding compatibility between Delphi and C++. The next stone on the path is laid by C++Builder's advantageous use of the Delphi VCL.

C++Builder uses the Delphi VCL in all of it's standard projects. It therefore needs to understand Delphi's types, objects and syntax. This was made possible in part by the fact that Object Pascal is similar in structure to C++, but supports a subset of the features available in C++. (There are some features of Object Pascal, such as sets, not supported by C++, but these are relatively minor roadblocks that can be overcome through a judicious use of templates and classes.)

The converse of the logic in the last paragraph is not true. There is no need for Delphi to parse C++. Furthermore, Object Pascal is, in a sense, a subset of C++. It is therefore easier for C++Builder to adopt to the features of Delphi, than it is for Delphi to support certain esoteric features of C++ such as multiple inheritance, variable parameter lists, or function overloading.

Object Pascal is not radically different from C++ in the sense that BASIC is radically different from C. The two languages, Object Pascal and C++, share the same structure.

C++Builder and Delphi share a common heritage in the way they construct objects, handle variable types, and define methods. In particular, a Delphi object's VMT is identical in structure to a C++ object that uses single inheritance. Furthermore, Delphi supports all the calling conventions, such as CDECL, PASCAL, and STDCALL, that are used with C++ functions and methods. Therefore, there is no difference in the structure of a Delphi function or method and a C++ function or method. They look the same on the assembler level, so long as C++ is not using any "advanced features", such as function over loading or variable parameter lists. (Remember that by default, Pascal uses the fastcall calling convention, which means that it passes some parameters to functions in registers. This same convention is supported by C++ Builder, or you can explicitly use a different convention if you like.)

Finally, there is a direct parallel between most C++ types and the standard Delphi types. For instance, a C++ **int** is identical to a Delphi **Integer**. Other types, such as strings, are either identical, or very similar. For instance, a C++ **char * (LPSTR)** is identical to a Delphi **PChar**. Furthermore, a Delphi string is fully compatible with a **char ***, and the C++Builder **AnsiString** type is designed to be compatible with, and to mimic, a Delphi **string**. This parallel structure continues throughout the two languages, and includes

complex types such as structures (records) and arrays. (What C++ calls a union, Delphi calls a variable record.)

There are, however, some differences between the languages. For instance, Delphi does not support templates, C++Builder does not support sets, and each language implements exception handling and RTTI slightly differently. These differences are significant, but they do not represent an impenetrable barrier erected between Delphi and C++Builder.

In this section you have seen that Delphi and C++Builder both use the same compiler, and support a very similar syntax and style of programming. These are the key elements that make it possible to share code between C++Builder and Delphi.

Using COM to link Delphi Code into C++Builder

The last few sections have shown that it is trivial to link Delphi units and components directly into C++Builder. Another great way to share code between the two tools is via COM and OLE. You might want to use COM and OLE if you want to share your code not only between Delphi and C++Builder, but also between Delphi and Visual Basic, or other OLE aware tools such as Word, Excel, Visual C++, etc.

There are three key ways to link Delphi COM objects into C++Builder:

1) Use OLE Automation or Dual Interfaces
2) Use Delphi 97 to create ActiveX controls
3) Write raw COM or OLE code to link in Delphi objects

The first two methods shown above are supported automatically by C++Builder. The third method requires you to dig into the details of OLE.

One simple way to link in Delphi objects via COM is to use OLE Automation. C++Builder fully supports OLE automation, and it provides a simple technique for accessing the methods of an Automation Object.

Automation Objects can be accessed in one of two fashions: You can access them via a COM IDispatch interface, or via dual interfaces. C++Builder has built in support for IDispatch, via the CreateOleObject VCL function and the Variant template class. This is the simplest method for accessing Automation Objects, and it is the one you should use unless speed is an extremely important issue for you.

For instance, suppose you have a Delphi IDispatch interface that exports the following methods:

```
ITest = interface(IDispatch)
  ['{1746E520-E2D4-11CF-BD2F-0020AF0E5B81}']
  function Get_Value: Integer; safecall;
  procedure Set_Value(Value: Integer); safecall;
  function Get_Name: WideString; safecall;
  procedure Set_Name(const Value: WideString); safecall;
  procedure Prompt(const text: WideString); safecall;
  procedure VarTest(var v1, v2, v3: Variant); safecall;
  property Value: Integer read Get_Value write Set_Value;
  property Name: WideString read Get_Name write Set_Name;
end;
```

Assume further that the DLL that contains this interface is referenced in the registry in association with CLSID that has a ProgID called “TestLib.Test”.

Here is how you can retrieve the interface and call the Prompt method from inside C++Builder:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
  Variant V = CreateOleObject("TestLib.Test");
  V.OleProcedure("Prompt", "Sammy");
}
```

This code first retrieves an instance of IDispatch inside a Variant. It then uses a method of the Variant class to call the Prompt method from the Delphi interface.

Here is how you would call the VarTest method:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Variant V1, V2, V3;

  Variant V = CreateOleObject("TestLib.Test");
  V1 = 5;
  V2 = "Sam";
  V3 = V;
  V.OleProcedure("VarTest", V1, V2, V3);
}
```

Delphi users looking at this code should note that Variant is a C++ class declared in SYSDEFS.H. It is not a simple type as it is when used inside Delphi.

If you want to use dual interfaces in C++Builder, then you can get very fast access to Automation Objects that reside in a DLL. You can also access local servers, that is Automation Objects in an executable, more quickly via dual interfaces, but the built in overhead with local servers is so great that the improvement in speed given by dual interfaces is less noticeable.

There are no built in tools for accessing dual interfaces in C++Builder. Obviously it will be possible to use them (after all, you can do anything in C++), but you will have to lay the ground work yourself. (The basic technique is very similar to the one outlined below in the section titled “Mapping Virtual Method Tables”.)

Using Delphi ActiveX Controls in C++Builder

C++Builder has built in support for ActiveX controls. You can load them into the environment and use them just as if they were native components. To get started:

1) Choose **"Component | Install"** from the menu.
2) In the **Install Components** dialog select the OCX button to bring up the **Import OLE Control** dialog.
3) All of the registered components on your system will appear in the **Import OLE Control** dialog. If you want to register a new component, select the Register button in the **Import OLE Control** dialog.
4) After selecting the control you want to import, press the OK button.
5) Your new control will now be listed at the bottom of the **Installed Components** dialog, and a interface source file will be placed in the LIB directory.
6) Press OK to recompile CMPLIB32 and add your control the Component Palette. By default, the new ActiveX control will appear in the OCX page of the Component Palette.

If you don't want to use the automated method shown above, you can also use the C++ language to build your own OLE containers. This is, of course, a difficult process, and the technique outlined above is usually infinitely preferable. Don’t forget that you can use the tools in Borland C++ to create OLE containers, and then link that code into your C++Builder project either directly, or through libraries, or through DLLs.

Using the Windows API to Link in Delphi COM and OLE Objects

Delphi 97 is an extremely powerful tool for creating COM and OLE objects. You can use it to build Automation objects, dual interfaces, ActiveX controls, and all manner of simple (or complex) COM objects.

C++Builder does not yet have the built in support for COM that you find in Delphi 97. However, it does have access to all the OLE code you find in Borland C++ 5.0, the Windows SDK, the MSDN, or in commercially available books and libraries.

The following is a definition for a simple Delphi COM object:

```
const
  CLSID_ITable: TGUID = (    D1:$58BDE140;D2:$88B9;D3:$11CF;D4:($BA,
$F3,$00,$80,$C7,$51,$52,$8B));

type
  ITable = class(IUnknown)
  private
    FRefCount: LongInt;
    FObjectDestroyed: TObjectDestroyed;
    Table: TTable;
  public
    constructor Create(ObjectDestroyed: TObjectDestroyed);
    destructor Destroy; override;
    function QueryInterface(const iid: TIID; var obj):
      HResult; override; stdcall;
    function AddRef: Longint; override; stdcall;
    function Release: Longint; override; stdcall;
  { interface }
    procedure Open; virtual; stdcall;
    procedure Close; virtual; stdcall;
    procedure SetDatabaseName(const Name: PChar); virtual; stdcall;
    procedure SetTableName(const Name: PChar); virtual; stdcall;
    procedure GetStrField(FieldName: PChar; Value: PChar);
      virtual; stdcall;
    procedure GetIntField(FieldName: PChar; var Value: Integer);
      virtual; stdcall;
    procedure GetClassName(Value: PChar); virtual; stdcall;
    procedure Next; virtual; stdcall;
    procedure Prior; virtual; stdcall;
    procedure First; virtual; stdcall;
    procedure Last; virtual; stdcall;
    function EOF: Bool; virtual; stdcall;
  end;
```

Here is the same interface as it would be declared inside a C++ project:

```
class IDBClass : public IUnknown
{
public:
   STDMETHOD(QueryInterface) (THIS_ REFIID, LPVOID*) PURE;
   STDMETHOD_(ULONG,AddRef)  (THIS) PURE;
   STDMETHOD_(ULONG,Release) (THIS) PURE;
   // Interface
   STDMETHOD_(VOID, Open) (THIS) PURE;
```

```
  STDMETHOD_(VOID, Close) (THIS) PURE;
  STDMETHOD_(VOID, SetDatabaseName) (THIS_ LPSTR Name) PURE;
  STDMETHOD_(VOID, SetTableName) (THIS_ LPSTR Name) PURE;
  STDMETHOD_(VOID, GetStrField) (THIS_ LPSTR FieldName, LPSTR Value)
    PURE;
  STDMETHOD_(VOID, GetIntField) (THIS_ LPSTR FieldName, int * Value)
    PURE;
  STDMETHOD_(VOID, GetClassName) (THIS_ LPSTR Name) PURE;
  STDMETHOD_(VOID,  Next) (THIS) PURE;
  STDMETHOD_(VOID,  Prior) (THIS) PURE;
  STDMETHOD_(VOID,  First) (THIS) PURE;
  STDMETHOD_(VOID,  Last) (THIS) PURE;
  STDMETHOD_(BOOL,  EOF) (THIS) PURE;
};
```

Given the above definition, you could write the following C++ code to use the Delphi object in C++:

```
#pragma argsused
void Window1_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT keyFlags)
{
 char CR[2] ="\r";
 HRESULT hr;
 PIDBClass P;
 LPSTR S, Temp;

 CoInitialize(NULL);
 hr = CoCreateInstance(CLSID_IDBClass, NULL, CLSCTX_INPROC_SERVER,
                       IID_IUnknown, (VOID**) &P);
 if (SUCCEEDED(hr))
 {
  S = (char *)malloc(1000);
  Temp = (LPSTR)malloc(100);
  P->SetDatabaseName("DBDEMOS");
  P->SetTableName("COUNTRY");
  P->Open();
  strcpy(S, "Countries and their capitals: ");
  strcat(S, CR);
  strcat(S, CR);
  while (!P->EOF())
  {
   P->GetStrField("NAME", Temp);
   strcat(S, Temp);
   strcat(S, ": ");
   P->GetStrField("CAPITAL", Temp);
   strcat(S, Temp);
   strcat(S, CR);
   P->Next();
  }
  MessageBox(hwnd, S, "COUNTRY TABLE", MB_OK);
  P->Release();
  free(S);
  free(Temp);
 }
}
```

Clearly this technique takes a bit of work, but it is not overly difficult. If you understand something about COM objects, it provides a viable method for sharing code between Delphi and C++.

Using DLLs to link Delphi Code into C++Builder

C ++Builder can easily access a Delphi DLL containing functions and procedures. If you want to access a Delphi object implemented inside a DLL from C++Builder then you should read the section below called Mapping Virtual Method tables.

If you have a Delphi DLL that you can call from inside a Delphi project then you do not need to change it at all to call it from inside C++ Builder. To get started, take the Delphi unit that lists the methods in your DLL and add it to your C++ project. C++Builder will automatically compile the unit and generate a C++ header file. In particular, you should link the unit in using the techniques described above in the section called "Ground Rules for Linking Delphi Code into C++Builder Projects."

You will not be able to access data declared in your DLL without first calling a function or procedure. This same limitation applies to all DLLs, regardless of the language used to implement them.

DLLs are useful when you have a large project that you want to divide into several modules. By placing code in DLLs, you can divide your projects into several binary files that can be loaded in and out of memory at will.

Mapping Virtual Method Tables

Delphi objects stored in a DLL are normally out of reach of Ebony projects. However, there is a way to get at them by matching the VMT of the Delphi object to the VMT of a virtual abstract, or "PURE", C++ object.

Consider the following Delphi declaration:

```
  TMyObject = class
    function AddOne(i: Integer): Integer; virtual; stdcall;
    function AddTwo(i: Integer): Integer; virtual; stdcall;
  end;
```

A virtual abstract version of the same object could be declared in C++ like this:

```
class __declspec(pascalimplementation) TMyObject: public TObject
{
public:
  virtual _stdcall int AddOne(int i) = 0;
  virtual _stdcall int AddTwo(int i) = 0;
};
```

To match up the VMT of the C++ object to the VMT of the Pascal object, all you need to do is retrieve a pointer to the object from the DLL. One way to do this is to export a function from the DLL which is designed for this explicit purpose:

```
function CreateObject(ID: Integer; var Obj): Boolean;
var
  M: TMyObject;
begin
  if ID = ID_MyObject then begin
    M := TMyObject.Create;
    Result := True
  end else begin
```

```
    M := nil;
    Result := False
  end;
  Pointer(Obj) := M;
end;

exports
  CreateObject name 'CreateObject';
```

You can call this method from the Ebony project with the following code:

```
typedef Boolean (_stdcall *LPCreateObject)(int ID, void *obj);

void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TMyObject *MyObject;
  LPCreateObject CreateObject;

  HANDLE hlib = LoadLibrary("OBJECTLIB.DLL");
  CreateObject = (LPCreateObject)GetProcAddress(hlib, "CreateObject");
  if (CreateObject(1, &MyObject))
  {
    int i = MyObject->AddOne(1);
    ShowMessage((AnsiString)i);
  }
  FreeLibrary(hlib);
}
```

The code shown here first declares a pointer to a function with the same signature as the CreateObject routine in the DLL. It then calls the standard Windows API functions LoadLibrary and GetProcAddress in order to retrieve a pointer to CreateObject. If the call succeeds, you can call CreateObject, return the address of the object you want to call, and then call one of it's methods.

Notice that all the methods in the Pascal object are declared virtual. This is necessary since you want to match up the VMTs, or Virtual Method Tables of the two declarations. If the methods weren't declared virtual, then there would be no virtual method table, and the ploy would not work. You can declare non-virtual methods in your object if you wish, but you will not be able to call these methods from C++.

Note also that the order of the methods you declare is very important. The names of the methods you want to call are not important to the C++ implementation; all that matters is the order in which the methods are declared.

You can use this same technique, or one like it, to export any object from a Delphi DLL into a C++Builder or Borland C++ project. You do not have to use GetProcAddress and LoadLibrary, but could instead import a Delphi unit that exports CreateObject directly into your C++Builder project. However, I use GetProcAddress in this example since it forces you to imagine the exact steps necessary to make this process work. In other words, it forces you to think about the addresses that are being shared between the DLL and the Ebony project.

Summary

In this paper you have learned that C++Builder has an unprecedented ability to access the code from it's sister product, Delphi. As a rule, you can simply link Delphi code directly into you C++Builder projects.

The types of code that can be shared between Delphi and C++Builder include forms, components, ActiveX controls, and simple methods and functions. You can simply link this code directly into your projects without any extra work.

This paper also examined using OLE or DLLs to share code between C++Builder and Delphi. OLE can be useful if you want to share code with not only C++ Builder, but also with other, non-Borland, environments, such as Word, Visual Basic, or Excel. DLLs are a great way to share code if memory management issues are significant. In particular, you can load and unload DLLs from memory during the life of your project.