Application Development with Borland C++ and Delphi

A Technical paper for developers

Alain Tadros with additions by Eric Uber

Borlan

d

Table of Contents

The purpose of this document is to familiarize you with ways of using your existing C/C++ code in Delphi and your existing Delphi code in C++.

The 1990's have proved to be a booming decade in the PC industry. With the growth of the PC market comes the growth of demands for programs that run on them. With this demand comes the need for more developers, better tools and better platforms on which these programs run. With the release of new development tools such as Borland's Delphi, the decision to use it can be greatly influenced by the time and cost of porting existing code.

By now, there are millions of lines of C/C++ code in existence which equates to billions of dollars for development. It is usually impractical to port large bodies of code from one language to another. Furthermore, you may have already just completed a large port -- from Pascal to 16 bit Delphi and then to 32 bit Delphi. Or maybe you ported your DOS C code to Windows in C++. I suspect you are not about to throw it all away so you can use only one programming language or the other. The purpose of this document is to familiarize you with ways to use your existing C/C++ code in Delphi and your existing Delphi code in C++ to productively build applications.

Note: *This document assumes you are using Delphi 2.0 and Borland C++ 5.0 unless otherwise specified.*

Mechanisms for using C++ with Delphi and Delphi with C++

There are three ways to integrate C++ and Delphi code. The first is through the use of Dynamic Link Libraries (DLLs). By following the various rules described in this document, functions and objects made available in DLLs in one of these languages can be accessed by the other language. The second is through object (OBJ) files Delphi now supports the generation and linking of OBJ files. Again, there are restrictions and rules which must be followed to integrate OBJ files from one language into another. The third approach is to use the Common Object Model (COM) to expose your C++ code as an object or set of objects, registering this object or objects with OLE (more accurately, COM) and then directly using these objects in Delphi. You can similarly use Delphi COM objects from within C++. While this last approach is perhaps the most elegant and general solution, it requires a fair amount of understanding of the COM infrastructure and is well-documented in existing literature. For this reason, we will concentrate on the first two approaches in this article.

You should choose whichever mechanism is most suitable to your program's implementation. Knowledge of the pros and cons of using DLLs or OBJs will assist in your selection.

Benefits of DLLs:

1. Smaller EXE footprint: The DLL is compiled separately from your main EXE and linked in dynamically at run-time. Because of this, it does not increase the size of your calling EXE.
2. Easy to maintain: The code is encapsulated in its own module (the DLL). Updating this portion of the code no matter how large is as

simple as sending your customers the changed DLL. Of course, with DLLs, you do need to be concerned with versioning issues between the EXE and DLLs.

3. Speeds development time: A "build all" of your main program source won't include the DLL source as it is compiled and linked separately.
4. Shared Code means global revisions: Multiple applications can call the same DLL file. Fixing a bug in the DLL file fixes the same bug in all applications that use it.

Disadvantages of DLLs:

5. Another file to distribute: The calling process must be able to find the file. Also, more files usually means more documentation.
6. Slower startup: The DLL is loaded separately and mapped into the loading process' address space. This means another Kernel file-mapping object must be created and the DLL must be sought out. Once the application has loaded, however, DLLs don't appreciably affect performance.
7. More room for programmatic errors: More allowances and considerations must be made programmatically as the loading module must match the calling conventions of the DLL and prevent C++ name mangling, (this is also an issue with OBJ), and the location of the DLL must be known.

Benefits of statically linked OBJs:

8. Fast runtime execution: Because there are no external calls to make.
9. Fewer modules to distribute: Everything is already linked into one single executable

Disadvantages of OBJs:

10. Large executable: The modules are linked in statically.
11. Can't share among multiple EXEs: Statically linked code becomes part of whatever EXE linked it in. Therefore, if code could potentially be shared among multiple EXEs, depending on the size/performance tradeoff, a DLL might be more appropriate.
12. More room for programmatic errors: More allowances and considerations must be made progammatically as the OBJ routines must match the calling conventions of the routine import declarations in the executeable using it.

Whichever mechanism is best for you, some additional work is still required as described a little later in this document.

Using Calling Conventions to ensure protocols are compatible

Whether or not you're using DLLs or OBJs, it is important to ensure that the code written in one language is generated following the protocol of the other.

The calling convention used defines important low level aspects of a program's behavior. It determines the order in which parameters passed to functions are put on the stack. It determines if registers share the responsibility with the stack of holding passed parameters. It determines

who is responsible for maintaining the stack i.e., whether the calling function or the called function cleans up the stack. In the case of C++, modifiers can be used to specify whether or not the symbol names get mangled.

Failure to correctly match calling conventions will result in fatal program errors and/or erratic program behavior and results.

Windows exports functions in its Application Programming Interfaces (API) using a calling convention different from Delphi's default calling convention. Similarly, Delphi's default calling convention is different from C++'s default. Fortunately both Delphi and C++ provide directives which allow you to change the calling convention of a method or function at compile time.

The calling convention used for exported functions in the Win32 API is called stdcall. A method or function following the stdcall calling convention requires parameters to be passed to it on the stack in right to left order. The called routine is responsible for cleaning up the stack. In C++ use the directive _stdcall. In Delphi, use the directive StdCall.

The following declares the same exported function in C++ and Delphi using the stdcall calling convention:

```
extern "C" int  _stdcall _export TestStdCall();
function TestStdCall: integer; StdCall; export;
```

The default calling convention used by Borland C++ is called cdecl. A method or function following the cdecl calling convention, like stdcall, requires parameters to be passed to it on the stack in right to left order. The difference from stdcall is in that the calling routine is responsible for cleaning up the stack. This convention is unique in that it supports the passing of a variable number of parameters. This type of support is available as result of the right to left parameter ordering.

In C++ use the directive _cdecl to explicitly declare a routine as cdecl although this is redundant as _cdecl is the default. In Delphi use cdecl.

Note that although Delphi supports the cdecl calling convention in terms of parameter ordering and stack maintenance, Delphi does not support the passing of a variable number of parameters. This becomes an issue when you attempt to export functions written in C++ that accept a variable number of parameters for use in Delphi.

The following declares the same exported function in C++ and Delphi using the cdecl calling convention:

```
extern "C" int  _cdecl _export TestCDecl();
function TestCDecl: integer; CDecl; export;
```

The default calling convention used by 16 bit Delphi 1.0 is called pascal. 16 bit Windows used this calling convention to export most of its API functions. A method or function following the pascal calling convention requires parameters to be passed to it on the stack in left to right order. This ordering is opposite of that used with cdecl and stdcall. Like stdcall, the called routine is responsible for cleaning up the stack. In C++ use the directive _pascal to explicitly declare a routine to use the pascal calling

convention. In Delphi, use the directive Pascal. In 16 bit Delphi, the use of the Pascal directive is redundant as pascal is the default convention .

The following declares the same exported function in C++ and Delphi using the pascal calling convention:

```
extern "C" int  _pascal _export TestPascal();
function TestPascal: integer; Pascal; export;
```

The default calling convention used by Delphi 2.0 is called fastcall. A method or function following the fastcall calling convention passes the first 3 parameters (that fit) in CPU registers EAX, EDX and ECX. The remaining parameters (if any) are passed on the stack in order from left to right. The called routine is responsible for cleaning up the stack. In C++, use the directive _stdcall. In Delphi 2.0 use the directive Register, however because FastCall is the default, the use of the Register directive is redundant.

The following declares the same exported function in C++ and Delphi using the fastcall calling convention:

```
extern "C" int  _fastcall _export TestFastCall();
function TestFastCall: integer; Register; export;
```

Note: *C++ uses name-mangling. Delphi does not. Use extern "C" for all functions exported to disable name-mangling so that Delphi will be able to link to them. When compiling in C, rather than C++, extern "C" is not required. See the Borland C++ on-line help for more information on extern and name-mangling.*

Exporting C++ Library Routines

Suppose you have existing code written in C++ that you wish to access from your Delphi code. You can wrap your routines up in to a black box module and compile it into a DLL with minimal work. The easiest scenario is that of a function library. Maybe you have a robust set of date/time routines that exceed the capabilities of those already available in Delphi. You can call them from a Delphi program by creating DLL access functions which are exported following consistent calling conventions. If you wish, you can export the routines directly. The following code fragment directly exports a routine in a DLL compiled using Borland C++:

```
//declaration
extern "C" BOOL  _stdcall _export IsTodayFriday();


//definition
extern "C" BOOL  _stdcall _export IsTodayFriday() {
  return FALSE;
}
```

If the routine IsTodayFriday were part of a function library whose prototype you would rather avoid modifying, you could export an access function to call it instead:

```
//declaration
extern "C" BOOL _stdcall _export AccessIsTodayFriday();
```

```
//definition
extern "C" BOOL  _stdcall _export AccessIsTodayFriday () {
  return IsTodayFriday();
}
```

It is really up to you to decide to use access functions or directly export your existing library functions themselves. Either way you have to write some code. However, the amount of code is far less than what would be required to rewrite the routines in one language or another. Again note that Delphi supports functions with a fixed number of parameters only no matter which calling convention is specified.

Functions to Access and manipulate C++ objects within a DLL

You can create and export access functions from a DLL that operate on objects instantiated within the DLL. This is often necessary when a Delphi program wants to use a C++ object but the C++ object has implemented function overloading. Delphi does not support function overloading.

Suppose you have a C++ class called TMyObject which has 3 AddToField member functions. Each AddToField member function expects a different parameter type and is therefore overloaded. Each AddToField member function writes to the correct field in a table based on the data type of the passed parameter. Consider the following declaration for TMyObject:

```
class TMyObject {
public:
    TMyObject();
    virtual BOOL _stdcall AddToField(long);
    virtual BOOL _stdcall AddToField(char*);
    virtual BOOL _stdcall AddToField(BOOL);
};
```

Since Delphi does not support function overloading, the class cannot be used directly. You could however write access functions which manipulate an instance of TMyObject. Consider the following:

```
//declarations
extern "C" BOOL _stdcall _export AddToField_LI(long,TMyObject*
obj);
extern "C" BOOL _stdcall _export
AddToField_STR(char*,TMyObject* obj);
extern "C" BOOL _stdcall _export
AddToField_BOOL(BOOL,TMyObject* obj);

//definitions
BOOL _stdcall _export AddToField_LI(long liVal, TMyObject* obj) {
    if (obj != NULL)
        return obj->AddToField(liVal);
    else
        return FALSE;
}
```

```
It is really up to you to decide to use acce  BOOL _stdcall _export
AddToField_STR(char* sBuf,TMyObject* obj) {
```

```
    if (obj != NULL)
        return obj->AddToField(sBuf);
    else
        return FALSE;
}

BOOL _stdcall _export AddToField_BOOL(BOOL bTorF,TMyObject*
obj) {
    if (obj != NULL)
        return obj->AddToField(bTorF);
    else
        return FALSE;
}
```

Above, 3 access functions are exported. Each has a slightly different name and a different data type in the first required parameter. The body of these functions call the appropriate AddToField method of the TMyObject object passed as a pointer in the second parameter. The access functions act as a wrapper for the overloaded AddToField methods.

Given the previous example, you are again left with a decision: should you rewrite your classes to not overload methods or should you take the time to write access functions? Keep in mind that any of your programs that use your existing classes would have to be changed if you choose to modify the overloaded functions themselves.

Using a C++ Object Instance in a DLL from a Delphi EXE

Delphi 2.0 and BC++ use essentially the same 32-bit compiler back end. The front ends (parser and semantic analyzer for each language syntax) are different, however. Because the back end (optimizer and code emitter) is essentially the same for both products, you can compile and link .OBJ files built with each language. The products also share the same virtual table structure (referred to as the "vtable" in C++ and "virtual method table" or "VMT" in Object Pascal) so you access virtual methods from an object created by either language from the other (as long as you use single inheritance, as multiple inheritance is not supported in Delphi).

The vtable/VMT holds the addresses of an object's methods all the way up the class hierarchy. It is by this mechanism that the correct method gets executed even if it has been overridden by several sibling classes. Because both BC++ and Delphi share the same vtable/VMT structure, your objects can cross the language boundary so long as the methods are declared as virtual.

The layout and representation of the fields within a Delphi object are not necessarily compatible with the layout and representation of the member variables of the C++ object. For this reason, when using a C++ object within Delphi, you can only rely on the ability to call virtual functions -- not access data members directly. This kind of programming interface style using abstract interfaces (classes with only virtual functions and no data members) is a very powerful one and is the basis of the OLE/COM object model. It provides a clear, functional interface between two subsystems and hides all the underlying implementation (data and non-virtual functions). It is this model that you must use when using C++ code from within Delphi.

Of course, flat "C" functions are callable as well and appear as global procedures and functions in Delphi.

In order to use an object in Delphi instantiated from a class in C++, the actual instantiation must occur on the C++ side. Since heap-allocated C++ objects are created with the C++ operator new, and the new operator and the corresponding memory management routines are not compatible with Delphi's, you must wrap the object allocation and destruction with flat functions and expose these flat functions to Delphi.. One function is called from Delphi to instantiate the object instance. This function uses the C++ operator new then returns to your Delphi program a pointer to the created object. When you are done using the object from within Delphi, the pointer is passed back to C++ code through the second access function, which releases the object from memory using the C++ operator delete. Consider the following declarations in C++:

```
class TMyObject {
 public:
    TMyObject();
    virtual int _stdcall VTOpenTable(char* sTableName);
    virtual int _stdcall VTDeleteRecord(int iRecNo);
    virtual int _stdcall VTCloseTable();
 };
 extern "C" TMyObject* _stdcall _export InitObject();
 extern "C" void _stdcall _export UnInitObject(TMyobject*);
```

The class TMyObject is declared first. You may either use an existing definition of a C++ class with virtual functions, or you may create a wrapper class that exposes the functionality of your object through virtual methods. If you use an existing class definition, be aware that you can only access it's virtual functions and not its member data. Also, be sure that your class uses only single inheritance.

The two functions InitObject and UnInitObject are declared with the extern "C" directive in order to ensure that C++ name mangling doesn't alter the names of these functions this is necessary because Delphi doesn't understand name mangling. If these wrappers are to be contained within a C++ DLL (as opposed to being linked from an OBJ), you will need to export these functions with the _export keyword. Further, these functions and all virtual functions are declared using the _stdcall calling convention. The calling conventions must match on both the Delphi and C++ sides and the only way to guarantee this is to explictly specify them.

The definitions of the wrapper functions are provided below and we'll see the code for our object later on:

```
 //Definitions
 extern "C" TMyObject* _stdcall _export InitObject() {
     return new TMyObject;
 }

 extern "C" void _stdcall _export UnInitObject(TMyObject* obj) {
     delete obj;
 }
```

The InitObject function creates a TMyObject object using the new operator and returns this pointer. When your Delphi program is done using the

object as returned from InitObject, it gets released from memory by passing it to UnInitObject which simply calls the C++ delete operator.

Although this appears straightforward on the C++ side, the Delphi side of this is a little tricky. In order for Delphi to treat the pointer returned from InitObject as an object, a class type TMyObject must be created in Delphi with the same method declarations as the TMyObject used in the C++ DLL. The difference is on the Delphi side, the methods are declared as abstract. This tells the Delphi compiler that the body of the methods are declared external to the unit in this case in the C++ code. Here is the needed declaration in Delphi code:

```
 { Class types }
 type
   TMyObject = class
     function VTOpenTable(pcTableName: PChar): integer; virtual;
stdcall; abstract;
     function VTDeleteRecord(iRecNo: integer): integer; virtual;
stdcall; abstract;
     function VTCloseTable: integer; virtual; stdcall; abstract;
   end;
```

The calling conventions on the object's methods (stdcall) must match that specified in the C++ program. They are also declared virtual and abstract.

```
 { Imports the Object Instance Access Functions from the C++
TESTOBJ DLL
}
 function InitObject: TMyObject; stdcall; far; external 'testobj.dll'
         name 'InitObject';
 procedure UnInitObject(obj: TMyObject); stdcall; far; external
'testobj.dll'
         name 'UnInitObject';
```

The example uses the stdcall calling convention and the methods are virtual. Let's take a look at the complete source to both the Delphi and C++ sides of the picture. We'll assume we're compiling the C++ code into a DLL and importing this DLL into the Delphi application.

Here is the C++ code which is compiled into a DLL.

```
 #include <WINDOWS.H>

 // TMyObject Class Declaration
 class TMyObject {
 public:
    TMyObject();
    virtual int _stdcall VTOpenTable(char* sTableName);
    virtual int _stdcall VTDeleteRecord(int iRecNo);
    virtual int _stdcall VTCloseTable();
 };
```

```
 //TMyObject Constructor
 TMyObject::TMyObject() {
    MessageBox(NULL, "Constructor called!", "TMyObject
constructor",
MB_OK);
 }

 //VTOpenTable Method hypothetically would open a table.
 int _stdcall TMyObject::VTOpenTable(char* sTableName) {
    MessageBox(NULL, "Code to open table goes here.", sTableName,
MB_OK);
    return 0;
 }

 //VTDeleteRecord Method hypothetically would delete a record.
 int _stdcall TMyObject::VTDeleteRecord(int i) {
    MessageBox(NULL, "Code to delete record goes here.",
"VTDeleteRecord",
MB_OK);
    return i;
 }

 //VTCloseTable Method hypothetically would close a table.
 int _stdcall TMyObject::VTCloseTable() {
    MessageBox(NULL, "Code to close table goes here.",
"VTCloseTable",
MB_OK);
    return 0;
 }

 #ifdef __cplusplus
 extern "C" {
 #endif

 //Create Object Instance-Access Function
 TMyObject* _stdcall _export InitObject(){
 return new TMyObject; //Alloc TMyObject instance
 }

 //Release Object Instance-Access Function
 void _stdcall _export UnInitObject(TMyObject * obj) {
 delete obj; //Release TMyObject instance
 }

 #ifdef __cplusplus
 }
 #endif
```

Notice the extern "C" {} which is wrapped around the definitions of the InitObject and UnInitObject global functions. This is necessary to ensure that C++ name mangling is disabled so that these functions can be imported into the Delphi application. We assume that this C++ program is compiled into a DLL named "testobj.dll" but omit the details of doing this.

The Delphi application in this case is extremely simple. The GUI is simply a form with a single button. When you click the button, InitObject is called

which returns the object for use in your Delphi program. Each method is then called and the object gets released. The following is the Delphi PAS file whose code makes use of the C++ DLL:

```
 unit Testobj;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

 type
  TForm1 = class(TForm)
   Button1: TButton;
   procedure Button1Click(Sender: TObject);
   procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
   {Private declarations }
  public
   {Public declarations }
  end;

 var  Form1: TForm1;

 implementation

 {$R *.DFM}

 { Class type TMyObject mirrors the C++ declaration but with abstract
methods. }
 type
  TMyObject = class
   function VTOpenTable(pcTableName: PChar): integer; virtual;
stdcall; abstract;
   function VTDeleteRecord(iRecNo: integer): integer; virtual;
stdcall; abstract;
   function VTCloseTable: integer; virtual; stdcall; abstract;
  end;


 { Imports the Object Instance Access Functions from the C++
TESTOBJ DLL
}
 function InitObject: TMyObject; stdcall; far; external 'testobj.dll'
        name 'InitObject';
 procedure UnInitObject(obj: TMyObject); stdcall; far; external
'testobj.dll'
        name 'UnInitObject';

 { This program has a single control of type TButton. Here is its
OnClick }
 procedure TForm1.Button1Click(Sender: TObject);
 var  obj: TMyObject; { Declare TMyObject Object }
 begin
  obj := InitObject; { Get the TMyObject object by calling
                the function InitObject in the C++ DLL. }
```

```
  obj.VTOpenTable('PARTNO.DB'); { Call each TMyObject method  }
  obj.VTDeleteRecord(10);       { remember that the body of these }
  obj.VTCloseTable;             { methods are defined in the C++ DLL.}
  UnInitObject(obj);            { You're done so release the TMyObject
                                  object by passing it back to the
                                  C++ DLL via the UnInitObject access
                                  function. }
 end;

 { The Forms OnClose event }
 procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
 begin
  Action := caFree; { Release the form from memory. }
 end;

 end. {End Unit}
```

If you don't want to use a C++ DLL, but would rather link your C++ obj files directly into your Delphi application, you need to specify the C++ obj file with the following directive in the Delphi unit's implementation section.

```
 {$L myobj.obj}
```

In addition, instead of importing the C++ wrapper functions into the Delphi application, you must declare them as external (since as they are contained within an external OBJ file, they are external to the unit which uses them).

```
 function InitObject: TMyObject; stdcall; far; external;
 procedure UnInitObject(obj: TMyObject); stdcall; far; external;
```

By following some simple conventions and rules, you can easily integrate C++ code into your Delphi application. We've shown you a very simple example; the same principles, however, would apply to more complicated, real-world examples. In summary, here are the conventions and rules to keep in mind:

- Your global functions and (member functions) must use match ing calling conventions (stdcall was used in our example)
- Your classes must declare and define virtual functions in C++ and only use single inheritance.
- Your classes must be declared in Delphi as well with virtual, abstract methods
- You must use extern "C" {} on the C++ side to disable name mangling for any functions imported into the Delphi application
- You must use wrapper functions to wrap the C++ new and delete operators. These are imported by declaring external on the Delphi side. The C++ objects must actually be created in C++ code.

Calling a Delphi DLL from a C++ executable

A DLL written in Delphi can be loaded and its exported functions called from an executable written in C++. Again note that the calling conventions for each function must be consistent across both languages. The following

shows the source code to a DLL called DDLL.DLL written using Delphi 2.0:

```
library DDLL;

uses Windows;

function GetDelphiString: PChar; StdCall;
begin
  MessageBox(0, 'Click OK and GetDelphiString will return a string!',
'Info',
            MB_OK or MB_TASKMODAL);
  result := PChar('This is a string passed from a Delphi DLL');
end;

exports
  GetDelphiString;

begin
end.
```

DLL.DLL exports a single functions called GetDelphiString. GetDelphiString Displays a message dialog box then returns a PChar string after the user clicks OK. A PChar is the C++ compliment of a NULL terminated character buffer. See the section at the end of this document called "Table of C++ and Delphi data types" for a listing of Delphi and C++ type comparisons. The following is the C++ code which uses DDLL.DLL and accesses the GetDelphiString function:

```
//Use IMPLIB.EXE against DDLL.DLL to generate a lib file which
//should be added to this example's project.
#include <WINDOWS.H>
#define IDC_PUSHBUTTON1 101

// C++ uses name-mangling.  Delphi does not.  Use extern "C" for all
functions
// exported from Delphi DLLs to indicate that the function is not
name-mangled.
// When compiling in C, rather than C++, extern "C" is not required.
See the
// Borland C++ on-line help for more information on extern "C" and
name-mangling.

extern "C" char*  _stdcall GetDelphiString();

// Globals
static HINSTANCE hInst;
char* StringPassed;

// This example assumes you have a Dialog resource called
"MAINDIALOG".

// The resource has 3 push buttons: IDOK,IDCANCEL and
IDC_PUSHBUTTON1.
```

```
 #pragma argsused
 LONG FAR PASCAL MainDialogProc(HWND hWnd, WORD wMsg,
WORDwParam, LONG lParam)
 {
   switch(wMsg)
   {   case WM_INITDIALOG:
          return TRUE;
       case WM_COMMAND:
          switch(wParam)
          {

          // Button was pushed, so call Delphi function
          case IDC_PUSHBUTTON1:
               StringPassed = GetDelphiString();
               MessageBox(NULL, StringPassed,  "Success", MB_OK |
MB_TASKMODAL);
               return TRUE;

          // Ok or cancel, so end the dialog
          case IDOK:
          case IDCANCEL:
                EndDialog(hWnd, 0);
                return TRUE;
          }
          break;
   }
  return FALSE;
 }

 //Program entry-has all the typical Windows stuff
 #pragma argsused
 int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                          LPSTR lpszCmdLine, int
nCmdShow)
 { // Save the instance.
   hInst = hInstance;

   // Load the dialog box.
   If ( DialogBox(hInstance, "MAINDIALOG", NULL,
(FARPROC)MainDialogProc)
== -1 )
        MessageBox(NULL, "Can't load dialog box!\n", NULL, MB_OK |
MB_APPLMODAL);
   return 0;
 }
```

The above executable simply prototypes the GetDelphiString function. The stdcall convention is used which matches the export from DDLL.DLL. The prototype uses extern "C" to prevent name mangling. Notice the return value is char*. This is a C++ compliment to Delphi's PChar type.

Using a Delphi Object Instance in a DLL from a C++ EXE

Earlier in this document ("Using a C++ Object Instance in a DLL from a Delphi EXE") the issue of using a class in a C++ DLL from a Delphi

executable was discussed. How about the reverse scenario? Using the C++ executable code sample above, make the following modifications:

```
//Use IMPLIB.EXE against DDLL.DLL to generate a lib file which
//Should be included in this example's project.
#include <WINDOWS.H>
#define IDC_PUSHBUTTON1 101

// C++ uses name-mangling.  Delphi does not.  Use extern "C" for all
functions
// exported from Delphi DLLs to indicate that the function is not
name-mangled.
// When compiling in C, rather than C++, extern "C" is not required.
See the
// Borland C++ on-line help for more information on extern "C" and
name-mangling.

class TMyObject   {
  public:
      virtual int _stdcall VTOpenTable(char* sTableName) = 0;
      virtual int _stdcall VTDeleteRecord(int iRecNo) = 0;
      virtual int _stdcall VTCloseTable() = 0;
};

extern "C" TMyObject* _stdcall InitObject();
extern "C" void _stdcall DeInitObject(TMyObject* oVertObj);

// Globals
static HINSTANCE hInst;
```

Really the only change thus far is in the removal of the declaration for GetDelphiString. It is instead replaced with the declaration of the class TMyObject. Notice the methods are all declared virtual and abstract. The calling convention chosen is stdcall. One more section of the code still requires modification as follows:

```
// Globals
static HINSTANCE hInst;

// This example assumes you have a Dialog resource called
"MAINDIALOG".

// The resource has 3 push buttons: IDOK,IDCANCEL and
IDC_PUSHBUTTON1.
#pragma argsused
LONG FAR PASCAL MainDialogProc(HWND hWnd, WORD wMsg,
WORDwParam, LONG lParam)
{
  switch(wMsg)
  {   case WM_INITDIALOG:
          return TRUE;
      case WM_COMMAND:
          switch(wParam)
          {
```

```
        // Button was pushed, so call Delphi function
        case IDC_PUSHBUTTON1:
        {
    TMyObject* obj = InitObject();
    obj->VTOpenTable("CUSTMAIN.DB");
    obj->VTDeleteRecord(10);
    obj->VTCloseTable();
    UnInitObject(obj);
          return TRUE;
        }
```

Here, the global declaration for char* StringPassed; was removed. Also, the code block following the case IDC_PUSHBUTTON1 has been modified. The new code block declares a pointer to an object of type TMyObject as it will be defined in the Delphi DLL. The InitObject function is called from the Delphi DLL which creates and returns an instance of TMyObject. Each TMyObject method is then called. The instance is released by passing it back to the Delphi DLL via the UnInitObject function. The following is the code for the Delphi DLL:

```
 library DDLL;

 uses Windows, Dialogs;

 type
  TMyObject = class
   function VTOpenTable(pcTableName: PChar): integer; virtual;
stdcall;
   function VTDeleteRecord(iRecNo: integer): integer; virtual;
stdcall;
   function VTCloseTable: integer; virtual; stdcall;
  end;

  function TMyObject.VTOpenTable(pcTableName: PChar): integer;
stdcall;

  begin
   ShowMessage('VTOpenTable');
  end;

  function TMyObject.VTDeleteRecord(iRecNo: integer): integer;
stdcall;
  begin
   ShowMessage('VTDeleteRecord');
  end;

  function TMyObject.VTCloseTable: integer;  stdcall;
  begin
   ShowMessage('VTCloseTable');
  end;

  function InitObject: TMyObject; StdCall;
  var oVertObj: TMyObject;
  begin
   oVertObj := TMyObject.Create;
```

```
  result := oVertObj
end;

procedure UnInitObject(oVertObj: TMyObject); StdCall;
begin
  oVertObj.Free;
end;


exports
  InitObject, UnInitObject;

begin
end.
```

The code for the Delphi DLL is rather simple. It exports the access functions InitObject and UnInitObject. These are declared using the directive StdCall The type section of the unit declares the actual TMyObject class. Its methods are all virtual following the stdcall calling convention. The actual method bodies don't do much, they simply acknowledge that they were actually called via a call to Delphi's ShowMessage function.

Note: *Once you build the Delphi DLL, you should re-run the IMPLIB.EXE utility so that the resulting .LIB file includes the InitObject and UnInitObject exports. The .LIB file should be compiled into the C++ executable.*

C++ OBJ linked into a Delphi Executable

The following example shows a C++ OBJ linked into a Delphi application. The code is simplistic in content as it exports a single function called COBJ_Function. The function does not accept any parameters and returns nothing. The function body simply calls the Windows API function MessageBox which will be displayed when called from the loading Delphi program. Here is the C++ source code:

```
// COBJ Example
// This is an example of an OBJ created with Borland C++ that is
linked

// into an EXE (DAPP.EXE) created with Delphi.
#include <WINDOWS.H>

//Declaration
extern "C"   void _stdcall COBJ_Function();

void _stdcall COBJ_Function()
{
  MessageBox(NULL,  "Hello from a Borland C++ OBJ!",
          "Success", MB_OK | MB_TASKMODAL);
  return;
}
```

The Delphi code is also rather simplistic. The following Delphi unit can be used in an application to link in the C++ OBJ and make use of the function COBJ_Function().

```
unit DAPPMAIN;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;

type
  TMain = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Main: TMain;

implementation

{$R *.DFM}

{Specify the name of the OBJ containing the function.}
{$L cobj.obj}

procedure COBJ_Function; StdCall; far; external;

procedure TMain.Button1Click(Sender: TObject);
begin
  COBJ_Function;
end;

end.
```

The compiler directive following the units implementation section really is what is of interest:

```
{$L cobj.obj}
```

Delphi doesn't have a pre-processor like in C++: therefore it makes use of compiler directives. To link a .OBJ file into a Delphi application, use the $L compiler directive followed by the name of the .OBJ file to bind in. To use a function in a .OBJ file, the function must be declared in Delphi as

external. Notice that the StdCall calling convention is used as well as far and external.

```
procedure COBJ_Function; StdCall; far; external;
```

The Delphi unit shown simply calls the COBJ_Funtion when the OnClick event for Button1 occurs.

Delphi OBJ Linked into a C++ Executable

The following example shows how to link in a Delphi OBJ into a C++ application. Here is the code for the C++ executable:

```
// CAPP Example
// This is an example of an EXE created with Borland C++ that calls a
// function that resides in an OBJ (DOBJ.OBJ) created with Delphi.
//Make sure you add DOBJ.OBJ to the project before linking.

#include <WINDOWS.H>;
#define IDC_PUSHBUTTON1 101


extern "C" {
      void _stdcall Delphi_Function();
}

static HINSTANCE hInst;

#pragma argsused
LONG FAR PASCAL MainDialogProc(HWND hWnd, WORD wMsg,
WORD
wParam, LONG lParam)
{
  switch(wMsg)
  {
      case WM_INITDIALOG:
          return TRUE;

      case WM_COMMAND:
          switch(wParam)
          {

          // button was pushed, so call Delphi function
          case IDC_PUSHBUTTON1:
               Delphi_Function();
               return TRUE;

          // ok or cancel, so end the dialog
          case IDOK:
          case IDCANCEL:
               EndDialog(hWnd, 0);
               return TRUE;
          }
          break;
  }
  return FALSE;
```

```
}

#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
{
  // save the instance
  hInst = hInstance;

  // load the dialog box
  if(DialogBox(hInstance, "MAINDIALOG", NULL,
(FARPROC)MainDialogProc) == -1 )
     MessageBox(NULL, "Can't load dialog box!\n", NULL,
MB_OK | MB_TASKMODAL);

  return 0;
}
```

The example executes the function Delphi_Function which is actually defined in an OBJ. file called DOBJ.OBJ. In order to use this function, it is declared at the top of the code listing as

```
extern "C" {
      void _stdcall Delphi_Function();
}
```

Note that extern "C" is used and _stdcall as the calling convention. When the programs dialog box window procedure receives IDC_PUSHBUTTON1 in the wParam of the WM_COMMAND message, the function Delpi_Function is called.

Take a look at the following code listing. It is the Delphi OBJ source code for the OBJ that gets linked into the C++ executable shown above. It's important to enable Project | Options | Linker | Generate Object files before compiling this example. (The default in Delphi is NOT to generate OBJ files.)

Note: *Many Windows API calls resolve to functions of other names. In C++, this is usually handled by the preprocessor. Since this Delphi code never goes through the C++ preprocessor, you must call the actual function name directly.*

```
unit dobj;
{
DOBJ Example
This is an example of an OBJ created with Delphi that is linked into
an EXE created with Borland C++.  It's important to enable
Project | Options | Linker | Generate Object files.
}

interface
```

```
uses windows;

procedure Delphi_Function; StdCall;

implementation

procedure Delphi_Function; StdCall;
begin
{ Many Windows API calls resolve to functions of other names.  In
C++,
  this is usually handled by the preprocessor.  Since this Delphi code
  never goes through the C++ preprocessor, you must call the actual
  function name directly. Below, MessageBoxA is called instead of
  MessageBox.
}

 MessageBoxA(0, 'Hello from a Delphi OBJ!',
             'Success', MB_OK or MB_TASKMODAL);
end;

begin
end.
```

Questions and answers

Question: Can I use any RTL functions in either language to create usable OBJs?

Answer: Yes and no.

No, because all the RTL functions get linked during Link time, not compile time; it's going to be the other language's responsibility to resolve those functions.

Yes, because you can make an OBJ out of the RTL functions and include it in the project with your other OBJ file. This way, though, is unrealistic due to the size of OBJ created for the RTL functions. This will make your executable much, much larger, and is probably not a good choice.

Question: What utilities can I use to facilitate mixing the 2 languages?

Answer: Borland ships 2 utilities with its language products that can be very helpful in that area, IMPLIB and TDUMP.

IMPLIB is a utility which converts a DLL file into a LIB file. You can then plug this LIB file into your project immediately. For more explanation and to see the different arguments, just type implib at the command line and press Enter.

TDUMP is a great utility and you should consider it your friend. It will tell you everything you need to know about your DLL, LIB or EXE files from the internals stand point: name mangling, exports, imports, library definitions, code- and data-segments, memory layout, etc.. You can check for function-name mangling, and can find the internal names of functions you want to use. You can explore the differences between calling conventions, by watching the function names change every time you change the calling convention.

For more information on TDUMP and its arguments, type tdump at the command line, and press Enter.

Question: I bought a Delphi library that contains a function that returns a STRING, how can I use this function in C++ although I have no access to the Delphi code?

Answer: There is no immediate way to use the returned STRING value in C or C++. The only way is to create another Delphi DLL that will take the String result as a parameter and return a pCHAR which will be usable in C or C++ as a CHAR * .

Conclusion

In summary, Borland Delphi 2.0 and Borland C++ 5.0 use essentially the same compiler back end. This provides a useful level of compatibility on the lowest levels. Delphi applications can link in Borland C++ OBJ files and Borland C++ applications can link in Delphi OBJ files. Because the Virtual Table (called the vtable in C++, VMT in Delphi) format is the same, the methods of an object instantiated in a DLL written in one language (specific to Borland Delphi 2.0 or Borland C++ 5.0) can be accessed from another. The Borland Delphi and Borland C++ languages

provide directives which support the standard calling conventions Cdecl, Stdcall, Fast-call and Pascal.

Given the information in this document, you should now be able to conclude which method of code sharing is best for you. It is unfortunate that additional work is needed for all of the methods discussed, however, it is great that Borland provides these mechanisms as the additional work is minimal compared to that which would be required to completely re-write all of your existing code in one language or another.

Table of C++ and Delphi data types

Delphi	C/C++
ShortInt	short
Byte	BYTE
char	unsigned short
Integer	int
Word	unsigned int
LongInt	long
Comp	unsigned long
Single	float
Real	None
Double	double
Extended	long double
Char	char
String	None
pChar	char
Bool	bool
Boolean	Any 1byte type
Variant	None
Currency	None