Integrating C++ Classes Into Delphi Applications

by Mzwanele

This article describes how to use C++ classes in your Delphi applications without having to know the intricacies of OLE 2.0 or how to write OCXs. You will be able to manipulate C++ objects as Delphi objects using properties, methods and events, or by creating Delphi components which use C++ proxy objects. All this comes courtesy of the Microsoft Component Object Model, or COM. Using these techniques I'll develop a Delphi component for the horizontal slider control from the Borland C++ Object Windows Library 2.5.

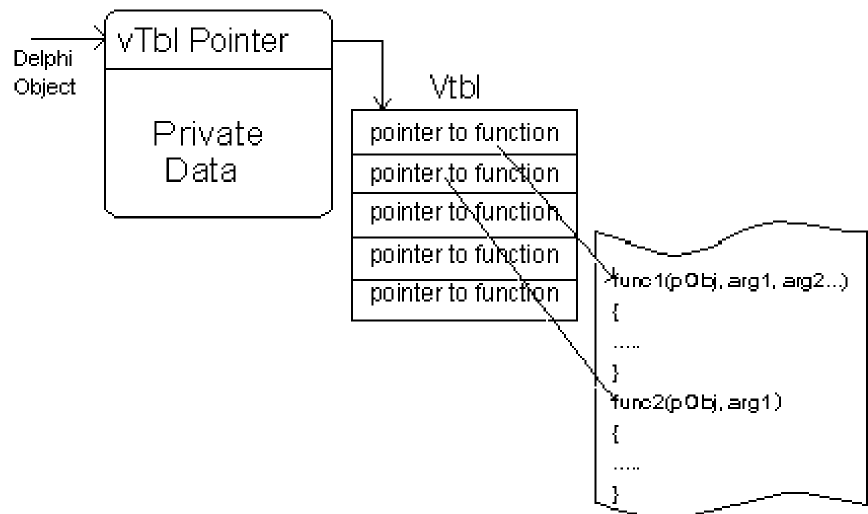
The Component Object Model

It seems that defining COM is nearly as difficult as defining the term object. The Microsoft view is that COM is a component architecture which gives users of this model the ability to develop applications built from components, supplied by different vendors and developed in different languages.

The component object is used as a building block in a larger system. Usually the component object has data associated with it (much like an OOP object), but the data should *not* be freely accessible outside of the component. Rather, access to the data should only be given through a published interface (the COM interface), using methods. COM therefore allows transparent access to components whether on the same system (across process boundaries) or on network processes.

The COM Interface

Access to a component object is achieved through a pointer, or rather a pointer to an array of pointers. A component object contains a pointer to a virtual function table (VTBL; or in Delphi parlance, a virtual method table), which holds pointers to the object's virtual methods. To access a specific member function, the pointer to the component object is de-referenced and a call is made through the VTBL to the method. The COM interface is really just a group of related functions through which the calling application can communicate with the component. Figure 1 shows an object's virtual method table layout. More extensive explanations of this model can be found in:



➤ *Figure 1: Object virtual method table layout.*

- *Delphi OLE Automation Servers*, by Danny Thorpe, from `http://www.borland.com/News/techlib/autosrv/autosrv.html`
- *Using Borland's Delphi and C++ Together*, by Alain Tadros and Eric Uber, from `http://www.borland.com/News/techlib/ brick.html`
- *The Revolutionary Guide to Delphi 2*, Chapter 17, WROX Press, ISBN 1-874416-67-2.

Using A C++ Class In Delphi

It should now be clear that COM can be used by any language which can call functions via pointers. Thus a C++ object can be used in Delphi, by putting it into a DLL and allowing the Delphi application to gain access to the C++ object pointer. On the Delphi side a class is created with an "empty" VTBL which has the same layout as the C++ class's VTBL, this class is then used to "control" the C++ object.

The process is as follows:

1. Declare a class in the C++ DLL, with exportable virtual methods.

2. Create an exported function which instantiates the C++ class dynamically. This function should also return a pointer to the C++ object. A second exported function is required which is used to destroy the C++ object once it is no longer needed.

3. Create a Delphi interface class with (`virtual abstract`) methods sequentially matching *all* the C++ class's virtual member functions (`private`, `protectedF255P255D and public`). The reason for this is to have the Delphi object's VTBL map directly onto the C++ object's VTBL. Thus every entry in the Delphi object's VTBL will have a corresponding entry in the C++

```
// In 16-bit:
// These #pragmas are only needed in the 16-bit version of the code
#pragma option -xc //Safe exceptions
#pragma option -WD //makes all functions in a DLL exportable
class _huge TDLLClass{
  char Buffer[80];
  int InternalValue;
  TEvent FEvent;
  virtual void  _pascal SetValue(int Info);
  virtual int   _pascal GetValue();
  virtual void  _pascal SetEvent(TEvent func);
public:
  TDLLClass();
  virtual void  _pascal ShowThevalue();
  virtual void  _pascal DoEvent();
};

// In 32-bit:
class  TDLLClass{
  char Buffer[80];
  int InternalValue;
  TEvent FEvent;
  virtual void  _pascal SetValue(int Info){InternalValue = Info;}
  virtual int   _pascal GetValue(){return InternalValue;}
  virtual void  _pascal SetEvent(TEvent func){FEvent = func;};
public:
  TDLLClass():InternalValue(0){FEvent.Code = NULL;};
  virtual void  _pascal ShowThevalue();
  virtual void  _pascal DoEvent();
};
```

➤ *Listing 1*

```
// In 16-Bit:
extern "C" {
TDLLClass* _pascal _export ConstructClass()
{
  return new TDLLClass;
};
void _pascal _export DestructClass(TDLLClass *DLLClass)
{
  if (DLLClass != NULL)
    delete DLLClass;
};
}

// In 32-Bit:
extern "C" {
TDLLClass* _cdecl _export ConstructClass()
{
  return new TDLLClass;
};
void _cdecl _export DestructClass(TDLLClass *DLLClass)
{
  if (DLLClass != NULL)
        delete DLLClass;
};
}
```

➤ *Listing 2*

```
{ In 16-bit: }
{ Mirror of the real C++ class - note the additional method attributes }
TDLLClass = class
  procedure SetValue(Info : integer);  virtual; abstract;
  function  GetValue : integer;        virtual; abstract;
  procedure SetEvent(func : TNotifyEvent); virtual; abstract;
public
  procedure ShowTheValue; virtual; abstract;
  procedure DoEvent;      virtual; abstract;
  property AnEvent : TNotifyEvent write SetEvent;
  property DLLValue: integer read GetValue write SetValue;
end;
{ In 32-bit: }
TCPPNotifyEvent = procedure(Sender: TObject) pascal of object;
TDLLClass = class
  procedure SetValue(Info : integer);  virtual; pascal; abstract;
  function  GetValue : integer;        virtual; pascal; abstract;
  procedure SetEvent(func : TCPPNotifyEvent); virtual; pascal; abstract;
public
  procedure ShowTheValue; virtual; pascal; abstract;
  procedure DoEvent;      virtual; pascal; abstract;
  property AnEvent : TCPPNotifyEvent   write SetEvent;
  property DLLValue: integer read GetValue write SetValue;
end;
```

➤ *Listing 3*

object's VTBL, creating a one to one relation between the Delphi VTBL and the C++ VTBL.

4. The Delphi EXE should import the C++ object creation and destruction functions from the DLL.

5. Implement Delphi event handlers in C++.

Well that's the theory! Let's look at a basic example to demonstrate the techniques.

Listing 1 shows the C++ class declaration required for Step 1 above. In 16-bit the C++ class must be declared as `_huge`; this modifier produces a far VTBL, allowing virtual methods to be called from outside the DLL.

The virtual methods used from the DLL all have to be exportable. In 16-bit use the `-WD` compiler switch to set all functions as exportable. In 32-bit all functions are exportable by default.

Now for Step 2, the exported C++ object creation and destruction functions. These are shown in Listing 2. The function that is exported from the DLL to give access to the C++ class is specified as `extern "C"`, instructing the compiler not to "mangle" the function name.

Although this example only demonstrates the creation of one object, further instances of C++ objects will be tracked and deleted by the DLL.

Listing 3 shows Step 3: the Delphi interface class. The Delphi interface class's methods which map to the C++ class's virtual functions are declared as `virtual abstract`, which means the methods are not implemented in the Delphi code but a slot is reserved for the function pointer in the VTBL. Thus the Delphi interface class's VTBL is identical to the VTBL of the C++ class.

The most noticeable difference between the Delphi and the C++ class is the use of properties in the Delphi class, which of course does not have an equivalent in C++. In Delphi properties are manipulated using accessor methods, or simply "get and set" functions:

```
property DLLValue: integer
  read GetValue
  write SetValue;
```

We can supply these "get and set" methods from within the C++ class, as shown in Listing 4.

Delphi 2.0 uses the fastcall (or register) calling convention by default, therefore you need to explicitly specify the calling convention of the virtual methods as `pascal` calling convention.

Now we move onto Step 4: importing the object creation and destruction functions into the Delphi EXE. For the Delphi code to be able to instantiate the C++ class in the DLL, it has to call a function in the DLL which must be imported by the Delphi EXE. In 16-bit:

```
function  ConstructClass :
  TDllClass; far;
  external 'CPPDLL';
```

and in 32-bit:

```
function  _ConstructClass :
  TDllClass;  cdecl
  external 'CPPDLL32';
```

Because the C++ object is created dynamically, we would like to deallocate the memory used by the object when we no longer have any need for the object. For this purpose we call an imported function in the C++ DLL to destroy the C++ class. In 16-bit:

```
procedure DestructClass(
  DLLClass :TDllClass);
  far; external 'CPPDLL';
```

and in 32-bit:

```
procedure _DestructClass(
  DLLClass : TDllClass);
  cdecl external 'CPPDLL32';
```

Finally, we need to implement event handlers in C++: Step 5. The event property will enable us to delegate some of the C++ object's behaviour to a Delphi object in the EXE. As a result, should an event occur in the C++ object, a Delphi object method will be called.

In general, calling a function using a pointer is not very difficult. However calling an object's methods using a pointer is more complex. This is because in most OOP and mixed-OOP languages a "hidden pointer" is passed to a method when it is called. This "hidden" pointer is the "this" or "Self" pointer of the object. It enables the function to access the data of that particular instance of the object. So, to be able to call an object's method via a pointer you actually need two pointers: one which points to the method and another one which points to the object instance. Delphi uses this technique to implement its delegation model.

Event properties or method pointers are referred to as "closures". A closure is a structure containing everything necessary to be able to call a method of an object.

Delphi uses the `TMethod` structure to implement closures. This structure contains two pointers: one pointing to the address of the class method and a data pointer which points to the instance of the object. `TMethod` is defined in SYSUTILS.PAS:

```
TMethod = record
  Code, Data: Pointer;
end;
```

To implement event handlers in C++ the `TEvent` structure is used, which is the equivalent of the `TMethod` structure in Delphi – see Listing 5.

To call a Delphi event handler in C++ we use:

```
virtual void _export _pascal
  TDLLClass::DoEvent()
  {
    if (FEvent.Code != NULL)
    ((TNotifyEvent)FEvent.Code)
    ((const void *)this,
    FEvent.Self);
  }
```

The Delphi object's method is called by de-referencing the code pointer and passing the Delphi object's instance pointer as the *last* parameter of the function, because the Delphi method uses the Pascal

➤ *Listing 4*

```
{ In the Delphi class: }
{ In 16-bit: }
TDLLClass = class
  ...
  procedure SetValue(Info : integer); virtual; abstract;
  function  GetValue : integer;       virtual; abstract;
 ...
end;
{ In 32-bit: }
TDLLClass = class
  ...
  procedure SetValue(Info : integer); virtual; pascal; abstract;
  function  GetValue : integer;       virtual; pascal; abstract;
  ...
end;

// In the C++ class:
class TDLLClass{
  ...
  virtual void  _pascal SetValue(int Info);
  virtual int   _pascal GetValue();
  ...
};
```

➤ *Listing 5*

```
typedef void _pascal (*TNotifyEvent)(const void *Sender,
  const void *thisPtr);
struct TEvent
 {
  void *Code;
  void *Self;
};
class _huge TDLLClass{
  ...
  TEvent FEvent;
  ...
};
```

calling convention (if the method used the C calling convention the instance pointer would be passed as the first parameter).

The `SetEvent` method of the C++ class is used in the assignment of the event property in Delphi. In C++:

```
class _huge TDLLClass{
  ...
  virtual void  _pascal
  SetEvent(TEvent func)
  {FEvent = func;};
  ...
};
```

and in Delphi:

```
TDLLClass = class
  ...
  property AnEvent :
    TNotifyEvent
   write SetEvent;
  ...
end;
procedure TForm1.Button4Click(
 Sender: TObject);
begin
  ACppClass.AnEvent := cProc;
end;
```

Because the Delphi method which is assigned to the C++ `TEvent` variable will be called from outside the Delphi EXE it must be exportable in the 16-bit version of the Delphi code.

When using event handlers in Delphi 2.0 the normal Delphi `TNotifyEvent` type cannot be used because of the fastcall calling convention used in Delphi 2.0, so we have to define a new method pointer type with the `pascal` calling convention:

```
TCPPNotifyEvent =
  procedure(Sender: TObject)
  pascal of object;
```

We have seen how easy it is to use a C++ COM class in Delphi code. Using this knowledge we can now continue further to create a more complex interaction between C++ and Delphi. We will create a Delphi component which uses a C++ Object Windows Library (OWL) control class: the OWL horizontal slider class. The slider control will be visible at design time and its properties can be changed through Delphi's Object Inspector.

OWL To Delphi In Two Easy Steps

Firstly we create a C++ COM class in a DLL, which will act as the OWL control's interface to the Delphi component. This C++ COM class will "carry" the OWL control by instantiating the OWL control and keeping a reference to the OWL control.

The OWL control is shown in Listing 6 and the C++ COM class in Listing 7.

Secondly, we create a Delphi COM interface class which maps to the C++ COM class. This Delphi class will in turn be "carried" by the Delphi component and used to communicate with the OWL control.

The Delphi COM interface is shown in Listing 8 and the Delphi Component in Listing 9.

In simpler terms, we use the C++/Delphi COM interface to simplify the interaction between a Delphi component and a C++ OWL control. On the surface we have a very clean and simple symmetry, as shown in Figure 2.

Windows Practicalities

Looking at this scenario, it seems as if every Windows message sent to the OWL control would have to be handled and the corresponding event handler in the Delphi component called. This would mean that an enormous amount of accessor methods and `TEvent` variables

Figure 2

OWL Control <– C++ COM <– DLL/EXE –> Delphi COM Interface <– Delphi Component

Listing 6: The OWL control

```
class TDHSLider : public THSlider, public MessageSink {
  public:
     TDHSLider(TWindow*        parent,
                     int              id,
                     int X, int Y, int W, int H,
                     TResId          thumbResId,
                     TModule*        module = 0):
                       THSlider(parent,id,X,Y,W,H,thumbResId, module),
                       MessageSink()
                       {
                             AddOWLClass(this);
                       };
  protected:
    void DoChange();
    virtual void SetPosition(int thumbPos);
    virtual LRESULT  Dispatch(TEventInfo& info, WPARAM wp, LPARAM lp = 0);
};
```

Listing 7: The C++ COM class

```
typedef void  (*TDispatchEvent)( const void *thisPtr, unsigned int Msg,
                                 unsigned int  wp, unsigned long lp);

class FARVTABLE TOWLDelphiControl {
     ...
     TWindow *InternalControl; //Reference to OWL control
  public:
    TOWLDelphiControl();
    ~TOWLDelphiControl();
     void InsertOWLControl(TWindow *IControl);
     unsigned long DoDispatch(uint Msg, WPARAM wp, LPARAM lp);
     virtual void   SetVisible(bool aValue);
     virtual bool   GetVisible();
     virtual void   SetEnabled(bool aValue);
     virtual bool   GetEnabled();
     virtual void   SetOnMessage(TEvent func);
     virtual void   BringToFront();
     ...
};
```

would have to be implemented in the C++ COM class. However, thankfully this is not the case.

Fortunately for us OWL is a well designed OOP application framework. Each OWL windowed control has a virtual method called `Dispatch`, through which all Windows messages sent to the OWL control will pass. By overriding this function each message can be passed on to the Delphi component through (you guessed it) a method pointer. The message is then passed onto the VCL by calling the Delphi component's `Perform`, method where it is turned into an event handler, which you use from Object Inspector. See Listing 10.

All that is left to do now is implement event handlers for any custom events we feel are needed, for instance an `OnChange` event.

➤ *Listing 8: The Delphi COM interface:*

```
TCMessageEvent = procedure (aMsg, wParam : Word; lParam: Longint)
  cdecl of object;
TCPPNotifyEvent = procedure (Sender : TObject) cdecl of object;
TCOMInterface = class
  procedure SetVisible(aValue: boolean); virtual; cdecl; abstract;
  function  GetVisible : boolean; virtual; cdecl; abstract;
  procedure SetEnabled(aValue: boolean); virtual; cdecl; abstract;
  function  GetEnabled : boolean; virtual; cdecl; abstract;
  procedure SetOnMessage(func : TCMessageEvent); virtual; cdecl; abstract;
  procedure BringToFront; virtual; cdecl; abstract;
  ...
end;
```

➤ *Listing 9: The Delphi component*

```
TDelOwlControl = class(TControl)
  private
    FOnMessage     : TMessageEvent;
    procedure FOnMessageExported(aMsg, wParam : Word; lParam: Longint);
      cdecl; export;
  protected
    OWLHelpControl : TCppDelphiControl; {Delphi COM interface}
    procedure VisibleChanging; override;
    procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy;    override;
    procedure  Invalidate; override;
    procedure  Repaint;    override;
    procedure  Update;     override;
  published
    property OnClick;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property Visible;
  end;
```

➤ *Listing 10*

```
\\This C++ code fragment shows how messages are passed to the
\\ Delphi component from the OWL control:
virtual LRESULT TDHSLider::Dispatch(
  TEventInfo& info, WPARAM wp, LPARAM lp = 0)
    {
       Dispatcher(info.Msg,wp,lp);
       return TEventHandler::Dispatch(info,wp,lp);
    };

{ In Delphi, we pass a message on to the VCL like this: }
procedure TDelOwlControl.FOnMessageExported(
  aMsg, wParam : Word; lParam: Longint);
begin
  if (csDesigning in ComponentState) or
     (csLoading in ComponentState) then
    Exit;
  Perform(aMsg, wParam, lParam);
end;
```

Dealing With Problem Parents

One last problem to contend with is the way OWL hooks into a non-OWL parent window's window procedure (`WndProc`) to create an OWL parent alias. This would normally not cause a problem, but in a RAD environment controls are created and destroyed as you design your application, which makes it impossible to safely create and destroy OWL parent aliases. To get around this problem all OWL parent aliases are kept track of and re-used as they are needed. These aliases are then only released at DLL unload time (a more efficient technique would be to use a reference count).

Using The Code Examples

This month's disk contains lots of code for you to experiment with. Three versions of the basic example are provided: one for 16-bit, one for 32-bit and one which can be compiled as both 16- and 32-bit. Each version has a Delphi project and a corresponding C++ DLL project compilable with Borland C++ 4.5.

For the OWL slider example, a C++ project for the OWL slider DLL is provided, as well as a Delphi project and Delphi component. Again, all are compilable as 16-bit or 32-bit.

To install the example OWL component provided on the disk follow these steps:

➢ Make a copy of your present COMPLIB.DCL (Delphi 1.0) or CMPLIB32.DCL (Delphi 2.0).

➢ Copy the OWLDEL.DLL (be sure to use the 16-bit DLL with the 16-bit Delphi EXE, the same goes for the 32-bit version) to your \WINDOWS or \WINDOWS\SYSTEM directory.

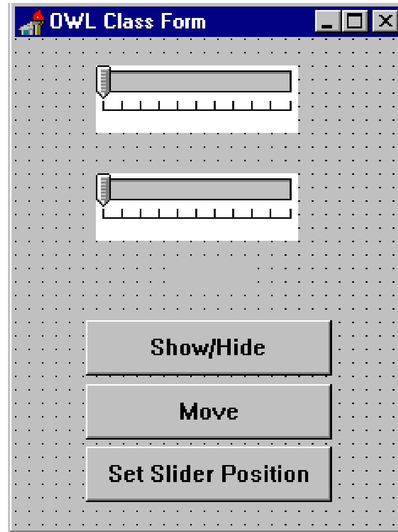
➢ Use the DELCOMPO.PAS to install the component; it will appear in the `Samples` palette page.

Conclusion

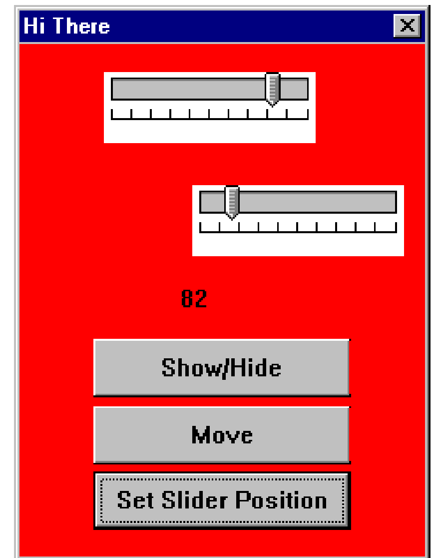
Well there you have it, I hope this will help you to integrate those painstakingly crafted C++ class libraries and help migrate those old C++ projects more easily.

Please be aware that all the techniques described in this article are not supported or guaranteed: you must use them at your own risk!

Mzwanele is a highly experienced Delphi developer working for an international software company. He can be contacted care of The Delphi Magazine.
Copyright 1996 Mzwanele



➤ *Figure 3: The Delphi form with OWL sliders at design time*



➤ *Figure 4: The Delphi form at run time with the OWL sliders*