Make C++ More "VB-Like"

Click & Retrieve Source CODE!

Combine the attributes of string handling with the functionality of a Windows edit box.

by Richard Hale Shaw

All right, I'll admit it: sometimes programming in C++, especially when using a C++ application framework such as the Microsoft Foundation Classes, can seem like a royal pain. After all, as easy as it is to build applications and components with VC++ and MFC, there are times when you wish you could write something as easily as you can with VB.

In this column, I'll describe some of the syntactic as well as structural differences of visual programming with Visual C++ and Visual Basic, and offer solutions that will let you write C++ code that looks and works like VB. In subsequent columns, I'll show you how to integrate these into an OLE control.

So how do Visual Basic and VC++ differ? Take edit controls, for instance. In both VC++ and in VB, you just drop them onto a form. VB immediately names each control (Text1, Text2, and so forth), and you can reference the code by name with code such as:

```
Text2.text = "This text goes into a VB _
   edit control."
```

Or to append a string to an edit control, how about:

```
Text2.text  = Text2.text + " And this _
   text goes in there, too."
```

Of course, VB doesn't have cool operators like C++'s += operator. Using the MFC CString class, which encapsulates string handling in almost every possible way, you can stuff some text into a string object:

```
CString myStr = "This text goes into an
   MFC CString object.";
```

And then append to the string with:

```
myStr += " And this text goes into it, too.";
```

There are several reasons for this kind of approach. First, C++ and C have a rich set of operators such as +=, which lets you combine the addition and assignment operators in such a way that, given that 'a' and 'b' are *ints*:

```
a = a + b;
```

and

```
a += b;
```

are equivalent.

Second, C++ is fully extensible and lets you create and add new data types to the language. You can create your own data types, customize them to make it easier to solve particular kinds of programming problems, and then derive even more specialized data types from the ones you already have.

Third, you can not only build collections of useful classes, but an integrated object hierarchy that's specifically geared toward solving the problems of Windows programming: that's what MFC is all about. And while MFC offers more than 150 data types or classes for Windows programming, it also includes useful helper classes such as CString. CString uses C++'s facility to *overload* or redefine the meaning of an operator in a given context, so that with CString, the += operator lets you *append* one string to another.

Of course, there's still the kind of problem that rears its head with Windows edit controls, especially when you use them in an environment like VC++. Let me explain.

VC++ AND EDIT CONTROLS

VC++ has its roots in traditional Windows programming, when virtually the only Windows programming language was C. In those days, a Windows program was carefully constructed from a plethora of different components: C source and header files, module definition (DEF) files, a make (MAK) file, a resource script (RC), a resource file (RES), and more. To add a button to a dialog, you had to run a standalone dialog editor to put a button on a dialog, and assign an ID—a unique integer value—to the button. The editor would modify a dialog resource (DLG) file, which would either have to be #included or integrated into the resource script.

The editor didn't assign the value for you: you had to choose a value and ensure that it didn't conflict with those in use by other controls in the same dialog. If you preferred to refer to the control through a macro or label (which is much cooler than using the number directly), you had to define the macro yourself and add it to a header file, where it'd be used by both the resource compiler (which compiled the RC into an RES), and by the C compiler (while compiling the C file). And that was how we did it in the old days.

Unfortunately, VC++ still suffers from this legacy. For example, when you drop an edit control onto a dialog template, VC++ assigns a not-so-useful label to the new control, like IDC_EDIT1, IDC_EDIT2, and so forth. These are the same kinds

Richard Hale Shaw is a contributing editor to Visual Basic Programmer's Journal *and* PC Magazine. *He's currently completing* Visual Programming++, *a book about Visual C++. He lives in Ann Arbor, Michigan, and can be reached on CompuServe at 72241,155, or the Internet at 3998368@mcimail.com.*

of labels we had in the old days, only now, VC++ generates them and assigns the integer value to them for you. This would be a nice feature—if it were 1990. But in 1996, with both VB and Delphi able to generate a variable that's associated with the control versus just a label, this is not exactly state of the art.

You can go a little further, of course. You can use the VC++ ClassWizard to create a dialog class data member and associate it with the control. For controls such as buttons, ClassWizard will create a new data member of type CButton, the MFC class that encapsulates the Windows BUTTON type. But for controls that have content, such as EDIT boxes, VC++ doesn't have a single approach to let you use an edit control interchangeably with an int, a string, or some other data type that has *content*.

For example, when you put an edit box into a dialog in an MFC application, VC++ gives you the option of creating not one, but two different kinds of class data members that can be mapped to the control. The one kind are "value" data members. These are data members such as int, long, float, or the MFC string type, CString; in other words, data members that have content (see Figure 1). The other kind are "control" data members. In the case of an edit control, the default choice is CEdit, the MFC class that subclasses a Windows edit control (see Figure 2). (You may recall that subclassing involves replacing the window procedure address of a window—or a control—with the address of another winproc that you provide. The purpose is to divert the window's message flow and intercept messages before the window can.)

First, the MFC developer has to decide how the control will be used. If you want to simply stuff some data into an edit control before the dialog box opens and then get the results out once it closes, you can use a value data member. This will allow you to initialize it before you open the dialog box (with CDialog::DoModal) and then get the resulting value out after the dialog closes. If you want to manipulate the control programmatically while the dialog is open, you can create a control data member: this is the best approach if you want to move data into and out of the control while the dialog box is open. But what if you want to do both?

You can still use a value data member when you plan to move data into and out of the control while the dialog is open. To do this, you can call the dialog's UpdateData member function (inherited by CDialog from CWnd). But CWnd::UpdateData will call the dialog's override of CWnd::DoDataExchange, which will exchange data between *all* of the dialog's controls and their associated value data members. That's fine if you want to update the controls or the data members *en masse*. For more granular control, however, it's the wrong approach.

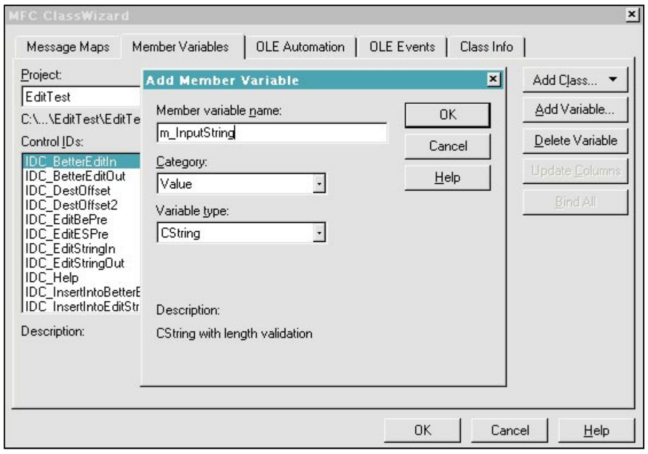
You can also use control data members to move data into or out of a particular control. For example:

```
CEdit myEditBox;
…
myEditBox.SetWindowText("This goes into
   the edit box…");
```

will use the string to replace the contents of the edit box subclassed by myEditBox. And:

```
CString myContents;
…
myEditBox.GetWindowText(myContents);
```

will retrieve the contents of the edit box into the CString object, myContents. This is the MFC equivalent of what you could



Project: EditTest
C:\...\EditTest\EditTe
Control IDs:
IDC_BetterEditIn
IDC_BetterEditOut
IDC_DestOffset
IDC_DestOffset2
IDC_EditBePre
IDC_EditESPre
IDC_EditStringIn
IDC_EditStringOut
IDC_Help
IDC_InsertIntoBetterE
IDC_InsertIntoEditStr
Description:

Add Member Variable
Member variable name: m_InputString
Category: Value
Variable type: CString
Description: CString with length validation

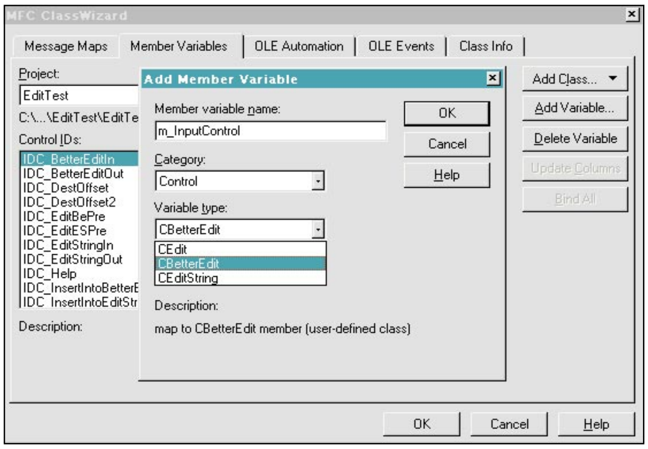
FIGURE 1 ***A Valuable "Value."*** *ClassWizard lets you create a "value" data member: a data member used to initialize the control before the dialog is opened, and to contain the control's final value after the dialog closes.*

accomplish in VB with:

```
myEditBox.text = "This goes into the edit box…"
```

or

```
dim myContents as string
myContents = myEditBox.text
```



Project: EditTest
C:\...\EditTest\EditTe
Control IDs:
IDC_BetterEditIn
IDC_BetterEditOut
IDC_DestOffset
IDC_DestOffset2
IDC_EditBePre
IDC_EditESPre
IDC_EditStringIn
IDC_EditStringOut
IDC_Help
IDC_InsertIntoBetterE
IDC_InsertIntoEditStr
Description:

Add Member Variable
Member variable name: m_InputControl
Category: Control
Variable type: CBetterEdit
CEdit
CBetterEdit
CEditString
Description: map to CBetterEdit member (user-defined class)

FIGURE 2 ***A Classy "Control."*** *ClassWizard also lets you create "control" data members that subclass the control, letting you intercept its message-flow and manage that in the control class.*

In both cases, MFC or VB is sending WM_SETTEXT or WM_GETTEXT messages to set or get the text stored in the control. It's just that VB offers the more elegant syntax.

The MFC approach of using a data type that subclasses the control will work just fine while the dialog is open. But once you close the dialog box (after DoModal returns), the relationship between the CEdit data member and the control it subclasses is finished. You can't use CEdit to retrieve the resulting text—the content of the edit box—after the dialog closes, because the edit box no longer exists.

The opposite is true as well. Until you open the dialog, the control doesn't exist for CEdit to subclass, so you can't stuff anything into the data member *before* the dialog is opened. (To be precise, the subclassing only takes place during the dialog's OnInitDialog override. CDialog::OnInitDialog calls CWnd::UpdateData, which calls the dialog's override of CWnd::DoDataExchange. This is where the subclassing takes place.)

The only thing that's even close to a built-in solution is to use *both* a string object as well as an edit control object. In other words, create two data members: one of type CString and one of type CEdit (meaning you use ClassWizard to create both a value data member and a control data member). Then you can use the CString data member to initialize the control before the dialog opens, and to retrieve the final value from the control after the dialog closes. And the CEdit data member will subclass the control, so you can use CWnd::GetWindowText and CWnd::SetWindowText—both inherited by CEdit, a CWnd derivative—to get and set the contents of the control while the dialog is open.

With Visual Basic, there's no distinction between a control and its content per se. You simply reference the control variable's Text property to get or set the data in the control.

To overcome MFC's lack of elegance and ease of use, I created a set of classes that lets you do with MFC what you can do with Visual Basic. I've presented one of these classes below, which shows you how to make C++ more "VB-like." In a future column, I'll show you a different solution, and how you can also use these when building an OLE control.

INTRODUCING THE CBETTEREDIT CLASS

You have a handful of fundamental problems to solve before creating a solution that combines the attributes of string handling with the attributes of a Windows edit box. First, I decided to produce an implementation that was a new data type. Creating a new C++ class provides a more efficient, flexible, maintainable, and extensible solution in the long run. For example, the new class could be extended to include new features in the future, as well as adapted to fit into environments that could really use it, such as OLE controls. So, I began with the idea of creating a new C++ class or data type.

Next, I had to figure out the best way to combine an edit control and string data handling into a single new class. MFC offers the CString as well as the CEdit classes, which both offered the features I was looking for and could produce the right result if combined properly. But what constituted "combining them properly"? My immediate reaction was, "Aha! A case for multiple inheritance." Let me explain.

If you're only familiar with VB 4.0's class facility, then you really don't know what object-oriented programming is all about. True, a single-level class like VB4's can let you combine relevant code and data into useful, reusable units. But the fact is, without inheritance you're missing out on the ability to easily extend a class for use in another context or situation. In C++, you can derive a new class from an existing one, where the new *derived* class inherits (with some minor restrictions and caveats) the features of the existing *base* class. You can even derive a new class from more than one base class: that's multiple inheritance.

C++ aficionados generally frown on MI (that's the cool acronym for multiple inheritance) because you can easily end up with a class that's far too complex to debug, maintain, or extend. It's an even bigger problem when one or more of the base classes are derived from a common base class. It's gotten to be such a hot topic that Bjarne Stroustrup, the inventor of C++, has noted that if he knew today what he knew about MI, he would never have added it to the language. And as my friend Zack Urlocker, Borland's Delphi product manager has said, "MI is the GoTo of the 90s."

So at first, I backed off of an MI solution, which would combine both CEdit and CString, even though these two didn't share any common base classes. (I later returned and reimplemented the solution using MI, but I'll discuss that in another column.) Instead, I tried to solve the problem using containment, the technique of having one type "contain" a data member of another type. This isn't as elegant and can take far more work if the contained item needs to get messages from Windows, but with this programming problem those weren't concerns. I created a new class, CBetterEdit, by deriving a class from CEdit and containing a CString data member (see Listing 1).

This approach involves using a CString to parallel and duplicate what Windows stores in the edit box's own data buffer. But note that I considered using a third approach: instead of containing a CString to duplicate the content of the control, I considered trying to play some games accessing the data buffer of the control itself. That data buffer can be obtained using Windows messages (EM_GETHANDLE, EM_SETHANDLE), and would require either containing a CString whose own data buffer pointed to the one in the control, or performing CString-like operations on the control's data buffer. With this approach, the big question was how Windows manages the data buffer of the control—something that's virtually undocumented in any version of Windows. Rather than monkey around with a potentially dangerous, nonportable solution, I decided that duplication wouldn't be bad for a starter: I could always re-engineer CBetterEdit to use that approach at a later date.

CBetterEdit gets its edit field capabilities from CEdit, from which it's derived. Its string-handling features come from the CString data member, m_strContents. The key to making CBetterEdit work was to create a few simple rules for myself: when you create the associated edit control, update it to contain the contents of m_strContents; when you destroy the associated edit control, update m_strContents to contain the contents of the control; when the user program makes changes to m_strContents, make sure they are reflected in the edit control; and, lastly, when the user makes changes to the edit control, make sure they are reflected in m_strContents.

These rules implied two explicit design goals: I'd have to intercept the WM_CREATE and WM_DESTROY messages when creating or destroying the associated edit control and follow the first two rules. Because CEdit subclasses its associated edit control, I simply created message-mapped member functions to override WM_CREATE and WM_DESTROY handling. These functions would ensure that I would meet my design goals.

You need to call the two message-mapped member functions, OnCreate and OnDestroy, right after you create or destroy the associated edit control:

```
int CBetterEdit::OnCreate(LPCREATESTRUCT
   lpCreateStruct)
{
   if (CEdit::OnCreate(lpCreateStruct)
      == -1)
      return -1;

   if(m_strContents.GetLength())
      SetWindowText(m_strContents);
   return 0;
}

void CBetterEdit::OnDestroy()
{
   GetWindowText(m_strContents);
   CEdit::OnDestroy();
}
```

```
class CBetterEdit : public CEdit
{
  void SyncContentsWithEditBox();
  void SyncEditBoxWithContents();

// Construction
public:
  CString Mid( int nFirst ) ;
  CString Mid( int nFirst, int nCount ) ;
  CString Right( int nCount ) ;
  CString Left( int nCount ) ;
  int GetNum();
  BOOL IsDigit();
  operator[](int nIndex) ;
  const CString& operator+=(LPCTSTR lpsz);
  const CString& operator=(char ch);
  const CString& operator+=(TCHAR ch);
  const CString& operator+=(const CString& string);
  int GetLength() ;
  operator LPCTSTR() ;
  const CString& operator=(LPCTSTR newContent);
  const CString& operator=(const CString&
newContent);
  LPCTSTR GetContents(void);
  void SetContents(char* newContents);
  CBetterEdit();

// Attributes
public:

// Operations
public:

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CBetterEdit)
  protected:
  virtual void PreSubclassWindow();
  virtual void PostNcDestroy();
  //}}AFX_VIRTUAL

// Implementation
public:
  virtual ~CBetterEdit();

  // Generated message map functions
protected:
  //{{AFX_MSG(CBetterEdit)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnDestroy();
  afx_msg void OnNcDestroy();
  //}}AFX_MSG

  DECLARE_MESSAGE_MAP()
private:
  CString m_strContents;
};
```

LISTING 1 ***Source Code for the CBetterEdit Class Definition.*** *Derived from MFC's CEdit class, CBetterEdit contains a CString data member that always contains the contents of the edit control associated with the class.*

OnCreate uses CWnd::SetWindowText (inherited from CWnd through CEdit) to update the edit control with the contents of m_strContents (provided there's anything in the latter). OnDestroy grabs whatever is in the control and stuffs it into m_strContents in the event that the user has changed the control's content just before closing the parent window or dialog (and the control is then destroyed).

Following my last two rules required more work.

SYNCHRONIZING THE CONTROL'S CONTENTS

I created two helper functions, SyncContentsWithEditBox and SyncEditBoxWithContents. These were both *private* member functions, to be called only from other member functions of this class (at a later date, I may make them *protected* so derived classes can use them). Both are implemented as inline functions, which tells the compiler to generate them as inline code. While this will result in slightly larger output code (because each call to these functions will be replaced by the function's code), it should result in code that will execute faster than if the function were actually called.

IF YOU'RE ONLY FAMILIAR WITH VB 4.0'S CLASS FACILITY, THEN YOU REALLY DON'T KNOW WHAT OBJECT-ORIENTED PROGRAMMING IS ALL ABOUT.

SyncContentsWithEditBox simply looks to see if there's an edit box associated with the class object and, if so, calls GetWindowText to get the edit box contents into m_strContents:

```
void SyncContentsWithEditBox()
   {
   if(m_hWnd)
      GetWindowText((CString&)m_strContents);
   }
```

SyncEditBoxWithContents does the opposite. It looks to see if there's a control associated with the object and, if so, stuffs the content of m_strContent into the edit box:

```
void SyncEditBoxWithContents()
   {
   if(m_hWnd)
      {
      CWnd *pWnd = (CWnd*)this;
      pWnd->SetWindowText((CString&)m_strContents);
      }
   }
```

By default, CString objects point to a NULL string ("") when you first create them or later empty them, so this is a safe way to empty an edit box when m_strContents has nothing in it. Using these two functions, I was able to implement functionality to let a user program employ this class to perform Basic-like string handling (through Mid, Left, and Right member functions); CString-like string handling (through +=, = and [ ] operator functions); and get or set the contents of the control through member functions (GetContents, SetContents member functions).

I even added a couple of new twists of my own: IsDigit to determine if the control contained an integer value, and GetNum to retrieve the control's content as an integer.

Using CBetterEdit, you can write code that smacks of the most elegant that MFC or VB has to offer such that:

```
CBetterEdit myEdit;
…
myEdit  = "This goes into the edit
   control";
```

will let you assign a string that's stored by the myEdit object, and additionally stuffed into the control once the control is created (or, if the control already exists, it is immediately stuffed into the control). You can use CString-like += operations as well. And you can use the assignment operator to update the control while it's displayed (without calling SetWindowText), ensure that the control will have an initial value once the dialog is opened, and easily get the final control value out after the dialog closes.

Check out the implementation of CBetterEdit and see what you think!