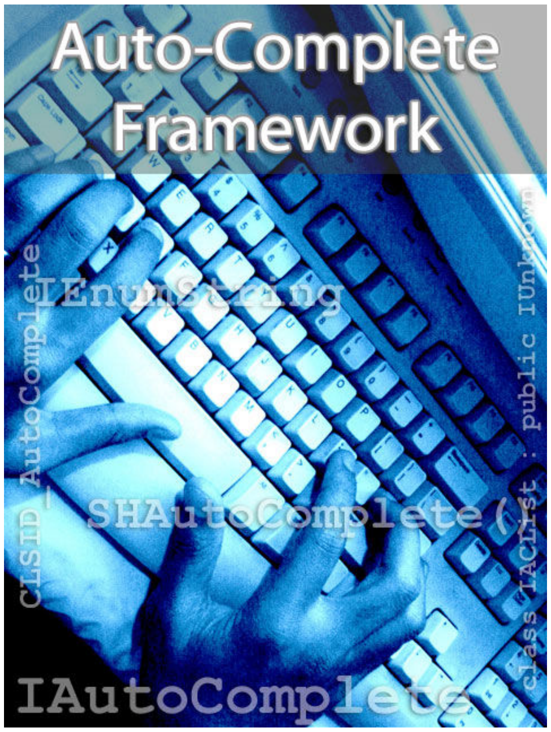
Using Microsoft's Auto-Completion Framework

By Remy Lebeau



In the July 2002 issue of the Journal, Damon Chandler showed how to write a custom `TEdit` descendant component that manually displays a popup `TListBox` which contains suggested string values that can be used to complete the partial string that the user has typed in [1]. In this article, I will describe a more robust and flexible way to accomplish the same thing, using a feature that has been built in to the Windows OS for several years now—the `IAutoComplete` interface.

What is auto-completion?

According to MSDN, auto-completion is defined as follows:

"Autocompletion expands strings that have been partially entered in an edit control into complete strings. For example, when a user starts to enter a URL in the Address edit control that is embedded in the Microsoft Internet Explorer toolbar, autocompletion expands the string into one or more complete URLs that are consistent with the existing partial string. A partial URL string such as "mic" might be expanded to "http://www.microsoft.com" or "http://www.microsoft.com/windows". Autocompletion is typically used with edit controls or with controls that have an embedded edit control such as the comboboxex control." [2]

MSDN also states that the `IAutoComplete` interface was introduced in SHELL32.DLL v5.0 and requires Windows 2000 or later. However, on a Windows 98 machine of mine, I have SHELL32.DLL v4.72 installed, and yet the auto-completion functionality is available and works. Looking in the Windows Registry, I found that the `IAutoComplete` interface (and related interfaces that will also be described in this article) is actually implemented in BROWSEUI.DLL v6.0. I do have Internet Explorer 6.0 installed on that machine, so it would appear that Microsoft moved its auto-completion system around in later versions of Internet Explorer and simply did not update their documentation accordingly. In any case, it is probably safe to assume that if a particular machine has Internet Explorer 5.0 or later installed, then the `IAutoComplete` interface is going to be available for your application to use. Internet Explorer 5.0 (or later) is preinstalled on all Windows 98/2000 and later systems, and can be downloaded for free for Windows 95/NT4.

Why use IAutoComplete?

Before Microsoft introduced the `IAutoComplete` interface, if an application wanted to have a UI that provided auto-completion options to the user, it would have to manually subclass an edit control in order to intercept the user's keystrokes, and it would have to do all of the management of the popup list-box and its contents—much like Damon demonstrated in his earlier article.

With the introduction of `IAutoComplete`, many of those details are now handled automatically for you by the Windows OS itself. Simply pass the `HWND` han-

Table 1: SHAutoComplete() options

Setting	Value	Description
`SHACF_AUTOSUGGEST_FORCE_ON`	0x10000000	Ignore the registry value and force the AutoSuggest feature on.
`SHACF_AUTOSUGGEST_FORCE_OFF`	0x20000000	Ignore the registry default and force the AutoSuggest feature off.
`SHACF_AUTOAPPEND_FORCE_ON`	0x40000000	Ignore the registry value and force the AutoAppend feature on.
`SHACF_AUTOAPPEND_FORCE_OFF`	0x80000000	Ignore the registry default and force the AutoAppend feature off.
`SHACF_DEFAULT`	0	The default setting, equivalent to `SHACF_FILESYSTEM | SHACF_URLALL`.
`SHACF_FILESYSTEM`	1	Include the file system.
`SHACF_URLHISTORY`	2	Include the URLs in the user's History list.
`SHACF_URLMRU`	4	Include the URLs in the user's Recently Used list.
`SHACF_URLALL`	6	Include the URLs in the users History and Recently Used lists. Equivalent to `SHACF_URLHISTORY | SHACF_URLMRU`.
`SHACF_USETAB`	8	Allow the user to select from the autosuggest list by pressing the TAB key.
`SHACF_FILESYS_ONLY`	16	Include the file system as in `SHACF_FILESYS_ONLY` plus directories, Universal Naming Convention (UNC) servers, and UNC server shares.
`SHACF_FILESYS_DIRS`	32	Indicates that the file system directories, UNC shares, and UNC servers should be enumerated.

dle of the desired edit control to `IAutoComplete` and everything is hooked up for you automatically. In addition, `IAutoComplete` offers some extra features that may be of interest to you:

- Suggested values can be populated with items from the File System, Internet Explorer's History and Favorites, the user's Recent Documents, the Shell Namespace, and even user-defined string lists. Any or all of these sources can be enabled at one time.
- Suggested values can be displayed using a popup list, or appended to the end of the edit control's current text, or both.
- A search engine can be invoked via a "Search" item that `IAutoComplete` can optionally append to each popup list that it displays.
- Quick completion of the user's typed-in partial strings via a hot key keystroke that invokes a pre-defined format string.
- RTL (Right-to-left) support for foreign systems, such as Hebrew and Arabic.

IAutoComplete in detail

Microsoft's auto-completion system is actually comprised of several ActiveX objects that work together. This allows you to pick-and-choose the features that you want to use in your application without taking up unnecessary resources.

The `IAutoComplete` interface works quite well for anything that exposes access to the `HWND` of a standard edit control, such as combo-box controls. The code provided with this article will show you how to wrap the main processing code into a generic, reusable class named `TAutoComplete`, as well as how to derive new components from `TEdit` and `TComboBox` to demonstrate how the class can be used.

NOTE: The Win32 API headers that Borland ships with each new IDE release tend to be dated. I will provide all relevant declarations in this article so that you can add them to your code if they are not already available in your version.

A first look at auto-completion

The simplest way to use auto-completion is to call the `SHAutoComplete()` function:

```
HRESULT STDAPICALLTYPE SHAutoComplete(
  HWND hwndEdit, DWORD dwFlags);
```

This function was introduced in SHLWAPI.DLL of Internet Explorer 5.0 and is declared in SHLWAPI.H. If you do not plan to do anything very complex with auto-completion in your code, it will probably suit your needs by itself, for example:

```
SHAutoComplete(hEditWnd, SHACF_FILESYSTEM);
```

This is a simple wrapper for the `IAutoComplete` interface and thus offers access to only a small subset of the available functionality. **Table 1** shows the available flag values at the time of this writing. I mention the function here only for the sake of completeness. For the rest of this article, I will focus on working with `IAutoComplete` and its related interfaces directly. If you are not familiar with working with ActiveX objects and interfaces, then I suggest you read up on that separately before continuing with this article.

Like other ActiveX objects, you have to obtain the `IAutoComplete` interface by calling the `CoCreateInstance()` function to instantiate an instance of its corresponding object, for example:

```
IAutoComplete *pAutoComplete;
::CoCreateInstance(
    CLSID_AutoComplete,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IAutoComplete,
    (LPVOID*) &pAutoComplete);
```

`CLSID_AutoComplete` and `IAutoComplete`, which are shown in **Listing A**, are declared in SHLGUID.H and SHLDISP.H, respectively. The other related interfaces, and their supporting flag values, are declared in SHLOBJ.H and OBJIDL.H. All you need to include in your own code is SHLOBJ.H, which includes all of the other header files.

NOTE: under C++Builder 5.0 or later, if you get "multiple declaration" errors from the shell header files during compiling, you will need to add `NO_WIN32_LEAN_AND_MEAN` to the Conditionals list of your Project Options in order for the code to compile properly. C++Builder 4.0 and earlier do not suffer from this problem. At the time of this writing, Borland has yet to fix this issue.

The Quick Completion feature

The `Init()` method of the `IAutoComplete` interface has two parameters that can specify the format string to be used for the "Quick Completion" feature. The format string behaves exactly like the format string of the C language runtime's `printf()` family of functions. When the user presses CTRL+ENTER in the edit control, the format string is applied to the edit control, where the current text is the value for the associated parameter of the format string. For example, if the format string is "http://www.%s.com", and the user types in "bcbjournal" and then presses CTRL+ENTER, the edit control will be updated to contain "http://www.bcbjournal.com" as its new value.

Listing A: IAutoComplete

```
DEFINE_GUID(CLSID_AutoComplete,
0x00BB2763L, 0x6A77, 0x11D0, 0xA5, 0x35,
0x00, 0xC0, 0x4F, 0xD7, 0xD0, 0x62);

MIDL_INTERFACE("00bb2762-6a77-11d0-a535-
00c04fd7d062")
IAutoComplete : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE Init(
    HWND hwndEdit, IUnknown *punkACL,
    LPCOLESTR pwszRegKeyPath,
    LPCOLESTR pwszQuickComplete) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    Enable(BOOL fEnable) = 0;
};
```

If the format string is stored in the Windows Registry, use the `pwszRegKeyPath` parameter of the `Init()` method, specifying both the key name and the value name. On the other hand, if the format string is in a null-terminated string in memory, then use the `pwszQuickComplete` parameter instead. For example:

```
// Registry
pAutoComplete->Init(hEditWnd, pUnk, NULL,
  L"Software\\MyKey\\MyFormatValue");

// Null-terminated string
pAutoComplete->Init(hEditWnd, pUnk,
  L"http://www.%s.com", NULL);
```

Auto completion sources

In order for the `IAutoComplete` interface to actually do anything, it needs to know the types of items that are to be displayed to the user. `IAutoComplete` itself does not generate the actual suggestions that the user sees. Other ActiveX objects that work in conjunction with `IAutoComplete` will provide the appropriate strings while the user is typing in the edit control. Any source that you use must implement the `IEnumString` interface, and optionally the `IACList` interface. Microsoft provides three ready-made objects:

1. `CLSID_ACLHistory` provides items from Internet Explorer's History list.
2. `CLSID_ACLMRU` provides items from the user's Recently Used Documents list.

3. `CLSID_ACListISF` provides items from the Shell namespace, including folders and files from the file system, as well as items from virtual folders such as My Computer and Control Panel.

Whenever `IAutoComplete` needs to provide the user with suggestions from a source, it enumerates through the `IEnumString` interface, shown in **Listing B**, to retrieve all of the available strings, which are then filtered as needed. `IACList`, shown in **Listing C**, is implemented whenever a source object can provide strings in a hierarchical manner (which all three of Microsoft's sources can). When the user begins typing into the edit control, only the matching items of the top level of the hierarchy are shown. When the user then types in a path delimiter, the current text of the edit control is passed to the `Expand()` method of `IACList`. This way, the source can update itself so that `IEnumString` will return the strings for the newly expanded level of the hierarchy from that point onwards. This approach allows `IAutoComplete` to efficiently show suggestions for very large hierarchies.

To use one of Microsoft's pre-made sources in your code, call `CoCreateInstance()` to instantiate an instance of its object, and then pass its `IUnknown` interface pointer to the `Init()` method of `IAutoComplete`; for example:

```
IACList *pIFS;
::CoCreateInstance(
  CLSID_ACListIFS, NULL,
  CLSCTX_INPROC_SERVER,
  IID_IACList, (LPVOID*) &pIFS);

pAutoComplete->Init(
  hEditWnd, pIFS, NULL, NULL);
```

You can implement your own ActiveX objects to provide custom string lists during auto-completion operations. The code provided with this article will show you how the `TAutoComplete` class implements some of its functionality using a custom ActiveX object to allow user-defined strings to be suggested.

Listing B: IEnumString

```
MIDL_INTERFACE("00000101-0000-0000-C000-
000000000046")
IEnumString : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE Next(
    ULONG celt, LPOLESTR *rgelt,
    ULONG *pceltFetched) = 0;
  virtual HRESULT STDMETHODCALLTYPE Skip(
    ULONG celt) = 0;
  virtual HRESULT STDMETHODCALLTYPE Reset()
    = 0;
  virtual HRESULT STDMETHODCALLTYPE Clone(
    IEnumString **ppenum) = 0;
};
```

Listing C: IACList

```
DEFINE_GUID(IID_IACList, 0x77A130B0L,
0x94FD, 0x11D0, 0xA5, 0x44, 0x00, 0xC0,
0x4F, 0xD7, 0xd0, 0x62);

class IACList : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE Expand(
    LPCOLESTR pszExpand) = 0;
};
```

Listing D: IObjMgr

```
DEFINE_GUID(CLSID_ACLMulti, 0x00BB2765L,
0x6A77, 0x11D0, 0xA5, 0x35, 0x00, 0xC0,
0x4F, 0xD7, 0xD0, 0x62);

DEFINE_GUID(IID_IObjMgr, 0x00BB2761L,
0x6A77, 0x11D0, 0xA5, 0x35, 0x00, 0xC0,
0x4F, 0xD7, 0xD0, 0x62);

class IObjMgr : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    Append(IUnknown *punk) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    Remove(IUnknown *punk) = 0;
};
```

Multiple auto-completion sources

The `IUnknown` interface for a single object that will provide all of the strings must be passed to the `Init()` method of `IAutoComplete`. If you are using only one source for your items, as shown in the example above, then you need only one object of the desired type. However, as I mentioned earlier, it is also possible to enable multiple sources at one time. **Listing D** shows another one of Microsoft's pre-made objects, `CLSID_ACLMulti`, which implements the

IObjMgr interface for this purpose.

In order to use multiple sources, you have to create an instance of the manager object and then call the Append() method of its IObjMgr interface for each source that you want to use. You can then pass the IUnknown interface pointer of IObjMgr to IAutoComplete, for example:

```
IACList *pMRU, *pIFS;
IObjMgr *pMgr;

::CoCreateInstance(
  CLSID_ACLMRU, NULL,
  CLSCTX_INPROC_SERVER, IID_IACList,
  (LPVOID*) &pMRU);
::CoCreateInstance(
  CLSID_ACListIFS, NULL,
  CLSCTX_INPROC_SERVER, IID_IACList,
  (LPVOID*) &pIFS);
::CoCreateInstance(
  CLSID_ACLMulti, NULL,
  CLSCTX_INPROC_SERVER, IID_IObjMgr,
  (LPVOID*) &pMgr);

pMgr->Append(pMRU);
pMgr->Append(pISF);

pAutoComplete->Init(
  hEditWnd, pMgr, NULL, NULL);
```

The IEnumString implementation of the CLSID_ACLMulti object will consolidate and merge the results of all the sources into a single list when needed by IAutoComplete.

The auto-suggest drop-down list

The IAutoCompleteDropDown interface, shown in **Listing E**, is obtained by calling QueryInterface() on the IAutoComplete interface. This interface allows the user to query the current state of the auto-suggest popup list, and to reset the current IEnumString enumerator while the auto-suggest list is visible; for example:

```
IAutoCompleteDropDown pDropDown;
DWORD dwFlags = 0;

pAutoComplete->QueryInterface(
  IID_IAutoCompleteDropDown,
  (LPVOID*) &pDropDown);
pDropDown->GetDropDownStatus(
  &dwFlags, NULL);

if( dwFlags & ACDD_VISIBLE )
pDropDown->ResetEnumerator();
```

Listing E: IAutoCompleteDropDown

```
MIDL_INTERFACE("3CD141F4-3C6A-11d2-BCAA-
00C04FD929DB")
IAutoCompleteDropDown : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    GetDropDownStatus(DWORD *pdwFlags,
      LPWSTR *ppwszString) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    ResetEnumerator() = 0;
};
```

Listing F: IAutoComplete2

```
MIDL_INTERFACE("EAC04BC0-3791-11d2-BB95-
0060977B464C")
IAutoComplete2 : public IAutoComplete
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    SetOptions(DWORD dwFlag) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    GetOptions(DWORD *pdwFlag) = 0;
};
```

Listing G: IACList2

```
DEFINE_GUID(IID_IACList2, 0x470141a0L,
0x5186, 0x11d2, 0xbb, 0xb6, 0x00, 0x60,
0x97, 0x7b, 0x46, 0x4c);

class IACList2 : public IACList
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    SetOptions(DWORD dwFlag) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    GetOptions(DWORD* pdwFlag) = 0;
};
```

Configuring auto-completion behavior

The rest of the available options for IAutoComplete are configured via the IAutoComplete2 and IACList2 interfaces, shown in **Listing F** and **Listing G**, which are obtained by calling the QueryInterface() method of IAutoComplete and IACList, respectively.

The SetOptions() method of IAutoComplete2 specifies how the results are displayed to the user, and how the edit control responds to certain user keystrokes; for example:

```
IAutoComplete2 *pAC2;
pAutoComplete->QueryInterface(
  IID_IAutoComplete2, (LPVOID*) &pAC2);

pAC2->SetOptions(ACO_AUTOSUGGEST);
```

The SetOptions() method of IACList2, on the other hand, specifies what kind of system resources are included in the results. The CLSID_ACListISF object is currently the only Microsoft object that uses IACList2, to specify what kind of file system items are to be used; for example:

```
IACList2 *pACL2;
pIFS->QueryInterface(
  IID_IACList2, (LPVOID*) &pACL2);

pACL2->SetOptions(ACLO_FILESYSDIRS);
```

The available flag values for both interfaces at the time of this writing are provided in **Table 2** and **Table 3** located at the end of this article.

Current working directories

The ACLO_CURRENTDIR option of the IACList2 interface requires some further explanation. Because the path of a file system item can be specified relative to a parent item, by using the ".." path notation, the IAutoComplete interface needs to be told what parent item to actually use when resolving relative paths from the user's input. IAutoComplete exposes two additional interfaces for this purpose.

The ICurrentWorkingDirectory interface, shown in **Listing H**, accepts the parent item as a Unicode string. Because a string is used, this interface is useful only for physical paths on the file system, for example:

```
pAC->QueryInterface(
  IID_ICurrentWorkingDirectory,
  (LPVOID*) &pCWD);

pCWD->SetDirectory(L"C:\\");
```

The IPersistFolder interface, shown in **Listing I**, accepts the parent item as an absolute ITEMIDLIST instead, relative to the root folder of the Shell namespace; for example:

```
IPersistFolder *pFolder;
LPITEMIDLIST pidl;
```

Listing H: ICurrentWorkingDirectory

```
DEFINE_GUID(IID_ICurrentWorkingDirectory,
0x91956d21L, 0x9276, 0x11d1, 0x92, 0x1a,
0x00, 0x60, 0x97, 0xdf, 0x5b, 0xd4);

class ICurrentWorkingDirectory :
  public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    GetDirectory(LPWSTR pwzPath,
      DWORD cchSize) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    SetDirectory(LPCWSTR pwzPath) = 0;
};
```

Listing I: IPersistFolder

```
DEFINE_GUID(IID_IPersistFolder,
0x000214EAL, 0x0000, 0x0000, 0xC0, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x46);

class IPersistFolder : public IPersist
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    Initialize(LPCITEMIDLIST pidl) = 0;
};
```

```
SHGetSpecialFolderLocation(
  NULL, CSIDL_CONTROLS, &pidl);
pAutoComplete->QueryInterface(
  IID_IPersistFolder, (LPVOID*) &pFolder);

pFolder->Initialize(pidl);
CoTaskMemFree(pidl);
```

IPersistFolder is more flexible then ICurrentWorkingDirectory because any item within the Shell namespace can be represented by an ITEMIDLIST. That includes not only physical paths on the file system, but also virtual folders, such as My Computer or the Control Panel, and items within virtual folders.

The TUserStringsEnum class

As I mentioned earlier, the IAutoComplete interface receives its string values from the IEnumString interface. By writing a custom ActiveX object that implements that interface, you can provide your own string values to IAutoComplete. The TUserStringsEnum class uses this approach to trigger OnExpand and On-

GetItem events in the TAutoComplete class. This way, users can implement their own storage for their custom strings. As you'll see in this month's code, TUserStringsEnum is a fairly straightforward class. It has the standard IUnknown methods for reference counting and interface querying, the standard IEnumXXX methods for enumerating strings, and the IACList methods for handling hierarchies.

TUserStringsEnum implementation notes

If you are familiar with writing custom ActiveX objects, one thing you will probably notice is that I am not deriving the TUserStringsEnum class from any pre-made ATL classes. There are four reasons for this.

First, IEnumString and IACList derive directly from IUnknown, and the ATL does not provide any wrapper class to handle the implementation for IUnknown itself, so it is not worth separating out that functionality into an additional class for the purpose of this article.

Second, because this class has a very simply design, and is private to TAutoComplete only, there is no need for the extra features like interface maps and the like that would have to be added to it to satisfy the ATL design model.

Third, although there are pre-made third-party IEnumString implementations available, they usually are implemented with STL containers that make copies of the string values for each object instance. For this class, I do not want to copy the user's strings around memory unnecessarily. Since there is no way to know where the user's strings are actually coming from, I want the TUserStringsEnum class to allow the user to optimize the access as desired.

Lastly, because the IEnumXXX interface design includes a Clone() method, TUserStringsEnum needs to be able to copy the enumeration values from one instance of itself to another. The purpose of cloning an enumerator is to maintain separate indices to the same source list. So there is no need to make copies of the user's strings in this situation either.

IEnumString implementation notes

The Next() method of the IEnumString interface is what actually retrieves one or more strings from the list. The calling code passes in an array of OLESTR pointers, and the number of strings needed. Next() is responsible for allocating a copy of the available strings, up to the number requested, and placing their memory pointers into the array.

Some people get confused when it comes to the memory allocation rules for ActiveX objects. In a nutshell, any method that returns data to the calling code must allocate the data, and then the calling code is responsible for freeing it later on. Unless otherwise indicated by the object's author, the data have to be allocated using ActiveX's own memory manager, which is the IMalloc interface, to ensure proper marshaling when the data pass across thread/process boundaries. For this article, the allocated data will not be crossing those boundaries; however, the ActiveX memory manager is still being used. I am using the CoTaskMemAlloc() function to keep the code simple, but the CoGetMalloc() function can instead be used to access the IMalloc interface directly, if desired.

The TAutoComplete class

Now that all of the options have been described, it is time to finally put the TAutoComplete class to work in your code. Upon examining the code that accompanies this article, you'll notice that the TAutoComplete class derives from TPersistent instead of TComponent. There, TAutoComplete is being embedded inside of other components, but you can use it as a standalone object if you desire.

By assigning an event handler to the OnGetItem and OnGetItemCount events, the TUserStringsEnum class will be used as one of the auto-completion sources (the Sources property specifies which, if any, of Microsoft's source objects will be used as well). While the user is typing in the edit control, IAutoComplete will call into the IACList and IEnumString interfaces of the TUserStringsEnum class, which will then trigger the OnExpand, OnGetItem, and OnGetItemCount events as needed.

With everything said and done, all that is needed to hook up the TAutoComplete class to your own code is the following:

1. Create an instance of TAutoComplete, such as in a constructor (and freed in the destructor).
2. Optionally set the Sources property. If you enable the acIFS flag, then set up the IFSOptions property as desired.
3. Assign OnGetItem and OnGetItemCount event

handlers if you wish to provide your own string values from an external source. Optionally, also assign an `OnExpand` event handler if your string values are organized in a hierarchy.

4. Keep the `EditHandle` property up-to-date.

Take special note of Step 4 above. VCL controls are notorious for destroying and recreating their window handles internally when their properties are changed during the life of the control. This can even happen multiple times. The `IAutoComplete` interface cannot do anything without a valid `HWND` assigned to it. If you use the `TAutoComplete` class inside of a `TWinControl`-derived component, then you should override the component's `CreateWnd()` and `DestroyWnd()` methods to keep the `TAutoComplete` object's `EditHandle` property in sync with the component's current `Handle` value.

Conclusions

Refer to the code that is provided with this article to see the complete implementation code. I hope you will find it useful for your projects.

Contact Remy at **remy@lebeausoftware.org**.

References

1. D. Chandler, "Auto-complete edit controls," *C++Builder Dev. Journal*, **6** (7), 2002.
2. Microsoft, "Using AutoComplete," http://tinyurl.com/l6xbh

Table 2: IAutoComplete2 options

Setting	Value	Description
`ACO_NONE`	0	Do not auto-complete.
`ACO_AUTOSUGGEST`	1	Enable the auto-suggest drop-down list.
`ACO_AUTOAPPEND`	2	Enable auto-append.
`ACO_SEARCH`	4	Add a search item to the list of completed strings. When the user selects this item, it launches a search engine.
`ACO_FILTERPREFIXES`	8	Do not match common prefixes, such as "www." or "http://".
`ACO_USETAB`	16	Use the TAB key to select an item from the drop-down list.
`ACO_UPDOWNKEYDROPSLIST`	32	Use the UP ARROW and DOWN ARROW keys to display the auto-suggest drop-down list.
`ACO_RTLREADING`	64	Normal windows display text left-to-right (LTR). Windows can be mirrored to display languages such as Hebrew or Arabic that read right-to-left (RTL). Normally, control text is displayed in the same direction as the text in its parent window. If ACO_RTLREADING is set, the text reads in the opposite direction from the text in the parent window.
`ACO_WORD_FILTER`	128	?
`ACO_NOPREFIXFILTERING`	256	Disable prefix filtering when displaying the auto-suggest dropdown. Always display all suggestions.

Table 3: IACList2 options

Setting	Value	Description
`ACLO_NONE`	0	Indicates that no enumeration should take place.
`ACLO_CURRENTDIR`	1	Indicates that the current directory should be enumerated.
`ACLO_MYCOMPUTER`	2	Indicates that My Computer should be enumerated.
`ACLO_DESKTOP`	4	Indicates that the Desktop Folder should be enumerated.
`ACLO_FAVORITES`	8	Indicates that the Favorites Folder should be enumerated.
`ACLO_FILESYSONLY`	16	Indicates that the file system should be enumerated.
`ACLO_FILESYSDIRS`	32	Indicates that the file system directories, UNC shares, and UNC servers should be enumerated.
`ACLO_VIRTUALNAMESPACE`	64	Indicates that the virtual namespace should be enumerated.