M. D. Wilson, "Safe CString Buffer Access"

The `CString` class is designed to fully-encapsulate a character buffer in an object, and provides many manipulation methods for dealing with the represented string, catering for most circumstances commonly encountered. Such classes in general, and the `CString` in particular, enable the C++ programmer to treat strings as first-class objects rather than C-style strings.

However, there are two circumstances in which treating strings as objects is not sufficient, both involving interactions with API functions that require C-style strings.

char const *

When string contents are required in non-modifiable form (which is the predominant case), they are usually specified as pointers to `const char` (or `const wchar_t`), as in

```
BOOL SetWindowText(HWND hwnd, TCHAR const *strText);
```

For this purpose, `CString` provides the `operator TCHAR const *() const` method - where `TCHAR` is a pre-processor symbol that is defined as `char` in ANSI compilations and as `wchar_t` in Unicode compilations - which simply returns a pointer to the start of the object's character buffer. Hence, the following code is legal

```
CString s(_T("the string"));
int i = strlen(s);
```

char *

On some occasions, access to modifiable character buffers is required, as in

```
DWORD GetCurrentDirectory(DWORD cch, TCHAR *s);
```

The `CString` class does not have an `operator TCHAR *()`, and even if it did, there could be problems - what would happen if the current space for the buffer is not enough for the API call to use? To address this, `CString` provides the `GetBuffer()` and `GetBufferSetLength()` methods with which the calling application may get a modifiable pointer (`TCHAR *`) to the object's internal character buffer. From the point of call of either of these functions the internal character buffer is notionally locked, and no other non-const member functions should be called. Once the calling code is finished with the internal character buffer, the method `ReleaseBuffer()` should be called, to indicate to the `String` instance that its internal buffer contents may have changed and to prompt it to re-establish the correct instance string length (via `strlen()`/`wcslen()`).

The problem

Unfortunately the use of these functions presents a number of potential problems.

Firstly, the idea of allowing client code to manipulate the internal workings of objects, over which the implementors of the object's class have no control, is a bad idea. Whatever the rationale for the provision and use of these methods, they are inherently unsafe and represent a danger. Whilst the technique provided here does not address this issue directly, it does help to limit the possibilities of the potential misuse of these direct-access methods.

Secondly, it is possible to forget to call `ReleaseBuffer()`, either by one of multiple return paths not making such a call or by functions called between the calls to `GetBuffer(SetLength)()` and `ReleaseBuffer()` throwing an exception. In this case the call to `ReleaseBuffer()` is lost, and the string contents may be corrupted.

An example demonstrating these problems is as follows:

```
CString str;
LPCTSTR psz = str.GetBuffer(_MAX_PATH);
LoadFieldFromDatabase(psz);
str.ReleaseBuffer();
WriteResult(str);
```

If `LoadFieldFromDatabase()` throws an exception, then the call to `ReleaseBuffer()` is lost.

The solution

The solution is in the form of the MFCSTL libraries' `grab_cstring_buffer` class. The class declaration is shown below. The full implementation is available from the MFCSTL web site.

```
// class grab_cstring_buffer
class grab_cstring_buffer
{
public:
    typedef grab_cstring_buffer   class_type;

// Construction
public:
    grab_cstring_buffer(CString &str, int length); // throw(CMemoryException *)
    ~grab_cstring_buffer() throw();

// Conversion operators
public:
    operator LPTSTR();
    operator LPCTSTR() const;

// Attributes
public:
    int length() const;
    int original_length() const;
```

```
// Members
protected:
    CString         &m_str;
    const int       m_len;
    const int       m_originalLen;
    const LPTSTR    m_psz;

// Not to be implemented
private:
    grab_cstring_buffer(class_type const &rhs);
    const grab_cstring_buffer &operator =(class_type const &rhs);
};
```

In its constructor, a reference to the `CString` instance is taken, the original length is remembered, and the modifiable character buffer pointer is obtained. If an exception is thrown in the call to `GetBuffer()` then the object is not constructed, and exception safety is preserved. Once the object is fully constructed, then any exception thrown by a called function will result in the destructor, and therefore `ReleaseBuffer()`, being called, so exception safety is preserved. Access to the buffer is via the `operator LPTSTR ()` method, and access to the original and requested lengths is via the `original_length()` and `length()` methods respectively.

There are three advantages to the use of this class.

- It is no longer possible that the call to `ReleaseBuffer()` can be forgotten, whether from simple omission or by having possible return paths from functions calling `GetBuffer(SetLength)()`. The `grab_cstring_buffer` class automatically calls `ReleaseBuffer()` on the `CString` instance in its destructor.

- The actual length of the `CString` instance's character buffer is lost as soon as the cycle is entered, and may or may not be the same as the requested length. The provision of the `length()` and `original_length()`enable access to reliable values of these attributes, which also facilitates the `grab_cstring_buffer` being passed in function arguments.

- Since the string should not be used during the `GetBuffer(SetLength)()` => `ReleaseBuffer()` cycle, it is safer to reduce the size of these cycles to a minimum. Using the `grab_cstring_buffer` class the previous example code block can be written as follows:

```
CString str;
LoadFieldFromDatabase(mfcstl::grab_cstring_buffer(str, _MAX_PATH));
WriteResult(str);
```

Since the `grab_cstring_buffer` class manipulates the `CString` instance in its destructor, the lifetime of the `CString` must encapsulate that of the `grab_cstring_buffer` instance. This usually presents no problem, since the `CString` instance can only be assigned to the `grab_cstring_buffer` instance in its constructor.

You should also be wary of writing code such as the following:

```
{
    CString                     str;
    mfcstl::grab_cstring_buffer gcsb(str, _MAX_PATH);

    LoadFieldFromDatabase(gcsb);

    WriteResult(str);
}
```

since the `gcsb` variable will not be destroyed prior to the call to `WriteResult()`, which means that it will not have called `ReleaseBuffer()` on `str`. Although calling const member functions on CString instances inside the GetBuffer(SetLength)() => ReleaseBuffer() cycle is legitimate, it would be all too easy to make a call to a non-const member, e.g. MakeUpper(), within this block of code. Hence the use of the inline form is preferred, as there are no such issues: the `grab_cstring_buffer` instances are temporaries, grabbing and releasing `CString` buffers and then disappearing before they can do any damage.

The text of this article originally referred to the Synesis Software `GrabCStringBuffer` class. This class was moved into the MFCSTL libraries in 2002, and this article updated accordingly.
