VC6, W2K, MFC
Posted 24 Dec 2003
Updated 27 Dec 2003
15,298 views

A class to make it easy to work with file names

By Rob Manderson

Working with filenames

21 members have rated this article. Result:
Popularity: 5.68. Rating: 4.3 out of 5.

Download source files - 4 Kb
Download demo project - 32 Kb

Introduction

As programmers we often find ourselves having to deal with filenames and to perform various operations on the filenames. Some years ago I found myself repeatedly writing code that received a filename and needed to perform some simple transformation such as changing the extension. After about the fifteenth function that contained code such as the following

```
TCHAR   tszDrive[_MAX_DRIVE],
        tszPath[_MAX_PATH],
        tszFilename[_MAX_FNAME],
        tszExtension[_MAX_EXT];
CString csNewPath;

_tsplitpath(tszFile, tszDrive, tszPath, tszFilename, tszExtension);
.
.
.
csNewPath = tszDrive;
csNewPath += tszPath + tszFilename;
csNewPath += someotherextension;
```

I realised that much of the work could be encapsulated in a class.

Background

You may recognise this class. It was first published on CodeGuru[^] but it's been extended since those days (August 1998) and a few bugs have been fixed. The class now understands Unicode.

The class encapsulates the basic idea of a filename whilst catering for UNC formatted filenames. It can also convert a filename specified, using drive letters on the local machine, into UNC format. How it does this depends on whether the drive portion of the filename refers to a local drive or a mapped network drive (I discuss this later in this article).

A filename has at least two forms. The first form is (for example)

```
c:\seti\01\seti.exe
```

You can break this down into four pieces.

- `c:` **is the hard drive name.**
- `\seti\01\` **is the directory path leading to the file.**
- `seti` **is the file name and**
- `.exe` **is the file extension.**

`CFileSpec` **understands this format.**

The other form that this class understands is a UNC filename of the form

```
\\Rob\C\seti\01\seti.exe
```

where

- `\\Rob\C` **is the server name and the share name.**
- `\seti\01\` **is the directory path leading to the file.**
- `seti` **is the file name and**
- `.exe` **is the file extension.**

You can see that apart from the server/share name or hard drive name, the rest of the filename spec is the same. The class leverages that equivalence by treating a drive name or a server/share name as interchangeable. But more on this equiavalence a little later...

The class interface

The class definition looks like this.

```
class CFileSpec
{
public:
    enum FS_BUILTINS
    {
        FS_EMPTY,       //  Nothing
        FS_APP,         //  Full application path and name
        FS_APPDIR,      //  Application folder
        FS_WINDIR,      //  Windows folder
        FS_SYSDIR,      //  System folder
        FS_TMPDIR,      //  Temporary folder
        FS_DESKTOP,     //  Desktop folder
        FS_FAVOURITES,  //  Favourites folder
        FS_MEDIA,       //  Default media folder
        FS_CURRDIR,     //  Current folder
        FS_TEMPNAME     //  Create a temporary name
    };

                    CFileSpec(FS_BUILTINS eSpec = FS_EMPTY);
                    CFileSpec(FS_BUILTINS eSpec, LPCTSTR szFileame);
                    CFileSpec(LPCTSTR szSpec, LPCTSTR szFilename);
                    CFileSpec(LPCTSTR szFilename);

//  Operations
    BOOL            Exists() const;
    BOOL            IsUNCPath() const;
    BOOL            LoadArchive(CObject *pObj) const;
    BOOL            SaveArchive(CObject *pObj) const;

//  Access functions
    CString&        Drive()         { return m_csDrive; }
    CString&        Path()          { return m_csPath; }
    CString&        FileName()     { return m_csFilename; }
    CString&        Extension()    { return m_csExtension; }
    const CString   FullPathNoExtension() const;
    const CString   GetFolder() const;
    const CString   GetFullSpec() const;
    const CString   GetFileName() const;
    const CString   ConvertToUNCPath() const;

    void            SetFullSpec(LPCTSTR szSpec);
    void            SetFullSpec(FS_BUILTINS eSpec = FS_EMPTY);
    void            SetFileName(LPCTSTR szSpec);

    void            Initialise(FS_BUILTINS eSpec);

private:
    BOOL            IsUNCPath(LPCTSTR szPath) const;
    void            WriteAble() const;
    void            ReadOnly() const;
    void            GetShellFolder(int iFolder);

    CString         m_csDrive,
                    m_csPath,
                    m_csFilename,
                    m_csExtension;
};
```

Let's ignore the `FS_BUILTINS` enum for now (and all constructors using that enum) and look at the constructors that take a string or two.

The simplest constructor takes a single string which is a filename and path. The constructor simply deconstructs the string into it's constituent parts via a call to `SetFullSpec` which in turn checks if the string is a filename in UNC format or not. If not it uses `_tsplitpath` to populate the `CStrings`. If it is a UNC filename it copies the server/sharename part of the string (if present) to the `m_csDrive` member and then uses `_tsplitpath` to deconstruct the remainder of the filename.

You can then use the member functions to retrieve the drive, the path, the filename, the extension, or the entire path. You can also change any of those constituent parts and expect the class to do reasonable things after the change. For example

```
CFileSpec fs(_T("c:\seti\01\seti.exe"));

fs.Extension() = _T("dat");
printf(_T("%s\n"), fs.GetFullSpec());
```

would print

```
c:\seti\01\seti.dat
```

showing that the extension was changed.

Note well that I said '(if present)' earlier. The class is perfectly capable of handing filenames that don't start at the top of the filename namespace and it doesn't care if it's a UNC path or not. (Though if it's a UNC path by definition it will be rooted at the top of the filename namespace).

So now we've got a filename parsed into it's component parts. Big deal. The power of the class comes from what you can do once you've got an instance of the class.

Suppose your application has an initialisation file stored in the same directory as the exe? How to get at it? You can use `GetModuleFileName()` to get the name of your exe and then do some massaging of the returned value to isolate the path portion of the returned filename and then append the initialisation filename to the result to get the final file. It might look somewhat like this.

```
TCHAR   tszDrive[_MAX_DRIVE],
        tszPath[_MAX_PATH],
        tszFile[_MAX_PATH];
CString csNewPath;

GetModuleFileName(NULL, tszFile, _MAX_PATH);
_tsplitpath(tszFile, tszDrive, tszPath, NULL, NULL);
csNewPath = tszDrive + tszPath;
csNewPath += _T("InitialisationFile.ext");
```

Or you could do this.

```
CFileSpec fs(CFileSpec::FS_APPDIR);

fs.SetFileNameEx(_T("InitialisationFile.ext");
LoadMyInitFile(fs);
```

or even this

```
CFileSpec fs(CFileSpec::FS_APPDIR, _T("InitialisationFile.ext"));

LoadMyInitFile(fs);
```

The same work is performed but you have to write rather less code.

Which leads into a discussion of the `FS_BUILTINS` that we ignored. These, via the constructors or the `SetFullSpec()` overload, let us initialise a `CFileSpec` object with the kinds of paths I thought were commonly used when I wrote the class. It's easy to extend the list if you think I missed something. (Thats a hint that if you want something added just add it, don't write a comment on this articles message board asking me to add it, write it yourself). That said, I'd be interested to know what things were added to `FS_BUILTINS` and would certainly consider adding them to the download.

If you examine the source code posted on CodeGuru 5 years ago you'll see that I used the Windows Registry to obtain the paths for the users desktop and favourites folders. Well it turns out that according to Raymond Chen[^], this is evil, so I changed the class to use the recommended method, `SHGetSpecialFolderLocation`. (I suspect Mike Dunn might also have railed at me if I'd continued using the registry :))

So far it's a pretty uninteresting though possibly useful class. Where it becomes interesting is in two features.

UNC Conversion

Elaine is working with an app that refers to data through drive `F:`. She doesn't know (and doesn't care) that her machine doesn't have a physical drive `F:`. Somehow, through the magic of drive mapping, a folder on the server called `\\Roger\CPData` is mapped to drive `F:` on her machine. Everything works as it should and, apart from network throughput issues, the world is a shining place. When Elaine closes her app it saves a reference to the drive it was using.

Somewhat later, Ian logs on to the same machine. He opens the same application and it, using the saved path, attempts to access the same files that Elaine was using. If Ian and Elaine have the same mapped drives it all works. But woe if Elaine mas mapped drive `F:` to `\\Roger\CPData` and Ian has mapped drive `F:` to `\\Stan\Rants`. This is where UNC conversion comes in. If Elaine's app had converted her reference to `F:/CPData` to a UNC path then, when Ian runs the same app under his ID it would refer to the same place, not caring if Elaine and Ian had the same drive mappings.

You call the member function `ConvertToUNCPath()` to perform the conversion. The function looks like this.

```
const CString CFileSpec::ConvertToUNCPath() const
{
    USES_CONVERSION;

    CString csPath = GetFullSpec();

    if (IsUNCPath(csPath))
        return csPath;

    if (csPath[1] == ':')
    {
        // Fully qualified pathname including a drive letter, check if it's a
        // mapped drive
        UINT uiDriveType = GetDriveType(m_csDrive);

        if (uiDriveType & DRIVE_REMOTE)
        {
            //  Yup - it's mapped so convert to a UNC path...
            TCHAR               tszTemp[_MAX_PATH];
            UNIVERSAL_NAME_INFO *uncName = (UNIVERSAL_NAME_INFO *) tszTemp;
            DWORD               dwSize = _MAX_PATH;
            DWORD               dwRet = WNetGetUniversalName(m_csDrive,
                                            REMOTE_NAME_INFO_LEVEL, uncName,
                                            &dwSize);
            CString             csDBShare;

            if (dwRet == NO_ERROR)
                return uncName->lpUniversalName + m_csPath + m_csFilename
                            + m_csExtension;
        }
        else
        {
            //  It's a local drive so search for a share to it...
            NET_API_STATUS   res;
            PSHARE_INFO_502 BufPtr,
                             p;
            DWORD            er = 0,
                             tr = 0,
                             resume = 0,
                             i;
            int              iBestMatch = 0;
            CString          csTemp,
                             csTempDrive,
                             csBestMatch;

            do
            {
                res = NetShareEnum(NULL, 502, (LPBYTE *) &BufPtr, DWORD(-1), &er, &tr,
                          &resume);

                //
                // If the call succeeds,
                //
                if (res == ERROR_SUCCESS || res == ERROR_MORE_DATA)
                {
                    csTempDrive = GetFolder();
                    csTempDrive.MakeLower();
                    p = BufPtr;

                    //
                    // Loop through the entries;
                    //
                    for (i = 1; i <= er; i++)
                    {
                        if (p->shi502_type == STYPE_DISKTREE)
                        {
                            csTemp = W2A((LPWSTR) p->shi502_path);
                            csTemp.MakeLower();

                            if (csTempDrive.Find(csTemp) == 0)
                            {
                                //  We found a match
                                if (iBestMatch < csTemp.GetLength())
                                {
                                    iBestMatch = csTemp.GetLength();
                                    csBestMatch = W2A((LPWSTR) p->shi502_netname);
                                }
                            }
                        }

                        p++;
                    }

                    //
                    // Free the allocated buffer.
                    //
                    NetApiBufferFree(BufPtr);

                    if (iBestMatch)
                    {
                        TCHAR tszComputerName[MAX_COMPUTERNAME_LENGTH + 1];
                        DWORD dwBufLen = countof(tszComputerName);

                        csTemp = GetFolder();
                        csTemp = csTemp.Right(csTemp.GetLength() - iBestMatch + 1);
                        GetComputerName(tszComputerName, &dwBufLen);
                        csPath.Format(_T("\\\\%s\\%s%s%s%s"), tszComputerName,
                                   csBestMatch, csTemp,
                                   m_csFilename, m_csExtension);
                    }
                }
                else
                    TRACE(_T("Error: %ld\n"), res);

                // Continue to call NetShareEnum while
                //  there are more entries.
                //
            } while (res == ERROR_MORE_DATA); // end do
        }
    }

    return csPath;
}
```

The function is `const` meaning that it doesn't modify the underlying `CFileSpec` object. It merely converts the path represented by that object into a UNC path (if possible). Either way it returns a `CString`.

The function first checks if the object already contains a UNC path. If so it returns the path.

If not, the function then checks if the object contains a path rooted to the file system namespace. Ie, does it contain a drive value. That's done by the line

```
if (csPath[1] == ':')
```

If this test fails we simply return the path the object represents (no conversion possible).

Assuming the test passes we then have two possibilities. The path may be on a share or it may be on local storage. The simpler case is when the path is on a share on another machine (ie, it's on a mapped drive). We get the drive type using the `GetDriveType()` API and branch to one of two paths. If it's a remote drive (the path is on a share) we execute this code.

```
//  Yup - it's mapped so convert to a UNC path...
TCHAR                tszTemp[_MAX_PATH];
UNIVERSAL_NAME_INFO *uncName = (UNIVERSAL_NAME_INFO *) tszTemp;
DWORD                dwSize = _MAX_PATH;
DWORD                dwRet = WNetGetUniversalName(m_csDrive, REMOTE_NAME_INFO_LEVEL,
                                 uncName, &dwSize);
CString              csDBShare;

if (dwRet == NO_ERROR)
    return uncName->lpUniversalName + m_csPath + m_csFilename + m_csExtension;
```

This is pretty simple. We call the `WNetGetUniversalName()` API passing the drive name (c:, d: etc) and, if the API succeeds, we get back a server/share name to which we append the remainder of the path. If the API fails (for whatever reason), the UNC conversion method returns the untranslated path.

Where it gets interesting is when the path refers to a drive on the local machine. In this case we want to search for a share on this machine that will be reachable from another machine.

To do this we need to enumerate shares. Naturally it's not quite that simple. Shares can be print queues, disk drives or IPC channels. Our code needs to ignore any share except drive shares. We do this by checking the share type for each share that's enumerated. Pretty simple. We set up the enum using the `NetShareEnum` API and then go into a loop examing each share returned. For each share that's a disk share we examine the path `p->shi502_path` associated with the share.

Now things get complicated. There may be more than one share on your machine that can reach the path. For example, you may have a share for `c:` and another share for `c:\seti`. Without considerations of permissions (and this code does no consideration) there's no necessarily best path. The code I wrote looks for the 'best match' where I define the 'best match' to be the longest server/share name. In other words, given a choice between `c:` and `c:\seti` and I'm looking for a match to `c:\seti\somefile` the code will choose the longest match. Of course, having chosen the longest match it has to delete some parts of the path.

This is done in this code

```
if (iBestMatch)
{
    TCHAR tszComputerName[MAX_COMPUTERNAME_LENGTH + 1];
    DWORD dwBufLen = countof(tszComputerName);

    csTemp = GetFolder();
    csTemp = csTemp.Right(csTemp.GetLength() - iBestMatch);
    GetComputerName(tszComputerName, &dwBufLen);
    csPath.Format(_T("\\\\%s\\%s%s%s%s"), tszComputerName, csBestMatch, csTemp,
                m_csFilename, m_csExtension);
}
```

We delete that part of the path that is contained in the share. Eg, if the share on machine `Rob`

is called `seti` and represents `c:\seti` on that machine, and the path is `c:\seti\setisrv.exe` then the returned UNC path will be `\\Rob\seti\setisrv.exe`.

A gotcha

If you've examined the code carefully you might be puzzled by this line.

```
csTemp = W2A((LPWSTR) p->shi502_path);
```

You'll have noticed that the function uses the `USES_CONVERSION` macro. This macro sets up some local variables to allow for Unicode to MBCS/ANSI conversion within the function (and vice versa). If you know about `USES_CONVERSION` then you probably also know about the `W2A` macro. This macro does a conversion from Wide to MBCS/ANSI, ie, from Unicode to MBCS/ANSI. The macro, if compiled within an ANSI/MBCS application, expects to see a Unicode input string (LPWSTR). Hmmm, so the example project included isn't Unicode. Why then the cast to `LPWSTR`? Well it turns out that in the version of the PSDK I have on this machine, the `SHARE_INFO_502` structure defines it's member variables as `LPSTR`'s if it's an ANSI/MBCS build and as `LPWSTR`'s if it's a Unicode build. Sounds good except for one thing. Regardless of how the build is done the OS returns Unicode strings! (Win2k SP4). If you've done an MBCS/ANSI build the code will return the wrong path unless you do the cast and use the `W2A` macro. So even though the compiler imagines that in an ANSI/MBCS build those members are ANSI/MBCS, they're really Unicode. But the `W2A` macro complains if you pass it a non `LPWSTR` argument. Hence the casts. This cost me a few minutes of head scratching.

Serialisation support

Lots of people don't like MFC's serialisation support. I've yet to see a convincing argument against it so this class contains support for it. This support wasn't a part of the original class. I just noticed that I was writing a lot of similar code so I added it and lost the need to write pretty much the same code for every new application I write.

```
open a file
create an archive object
attach the file to the archive
serialise the object through the archive
detach the file from the archive
close the archive
close the file
```

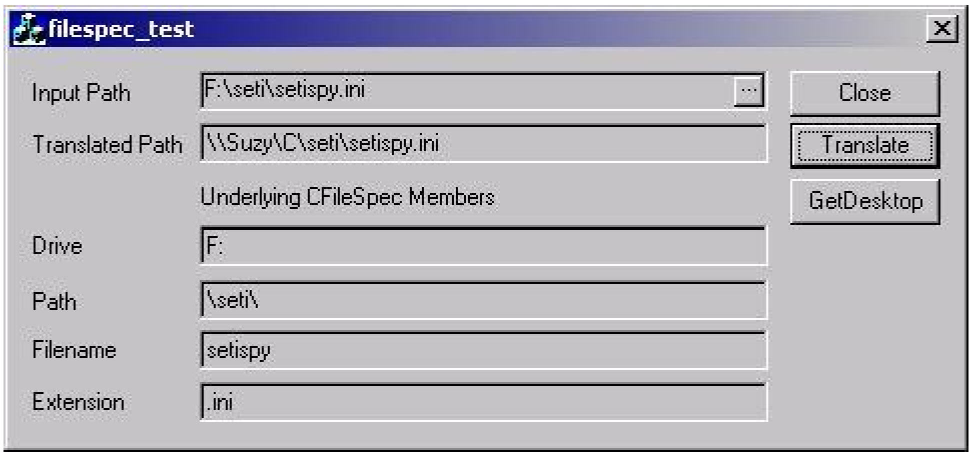
It seemed to me that adding MFC archive support to this class was a no brainer - and it was. You've seen the functions defined above in the class header. Here's the code.

```
BOOL CFilename::LoadArchive(CObject *pObj) const
{
    CFile file;
    BOOL  bStatus = FALSE;

    ASSERT(pObj);
    ASSERT_VALID(pObj);
    ASSERT_KINDOF(CObject, pObj);
    ASSERT(pObj->IsSerializable());

    if (Exists())
    {
        try
        {
            if (file.Open(GetFullSpec(), CFile::modeRead | CFile::typeBinary
                                 | CFile::shareExclusive))
            {
                CArchive ar(&file, CArchive::load);

                pObj->Serialize(ar);
                ar.Close();
                file.Close();
                bStatus = TRUE;
            }
        }
        catch(CException *e)
        {
            e->Delete();
        }
    }

    return bStatus;
}
```

Simple code. Check if the file exists. If it does, open it, attach an archive object to the file object and then serialize it into the object pointer passed. Naturally the object pointed to must support serialisation, hence the debug checks. It's almost exactly the same code for saving an archive.

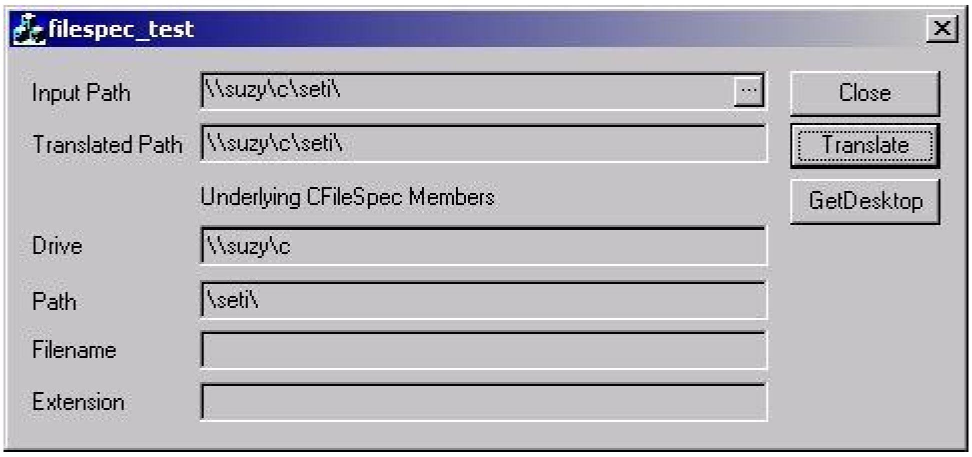
The Demo Application

The demo application does some simple conversions of paths to translated paths. It also shows the member variables of the underlying `CFileSpec` object. The examples shown here demonstrate how the class converts from a reference to my wifes machine (f:\seti\setispy.ini) where f: is mapped to \\Suzy\C.



Input Path: F:\seti\setispy.ini
Translated Path: \\Suzy\C\seti\setispy.ini
Underlying CFileSpec Members
Drive: F:
Path: \seti\
Filename: setispy
Extension: .ini

The second example shows how the class converts a UNC input path.



Input Path: \\suzy\c\seti\
Translated Path: \\suzy\c\seti\
Underlying CFileSpec Members
Drive: \\suzy\c
Path: \seti\
Filename:
Extension:

Acknowlegements

The sample project uses a class, `CFileEditCtrl`, written by P J Arends. An excellent class found here.

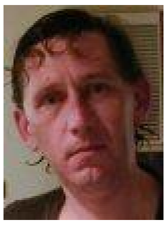
History

December 24 2003, Initial CodeProject version.

December 24 2003, Fixed a bug noted by PJ Arends.

December 27 2003, Fixed a bug noted by Mike Dunn.

About Rob Manderson



I've been programming for 27 years - started in machine language on the National Semiconductor SC/MP chip, moved via the 8080 to the Z80 - graduated through HP Rocky Mountain Basic and HPL - then to C and C++.

I wrote the software we use to run games at my online live trivia site - the game software and the game client are both written in MFC. Give the site a try - you might like it.

Born and bred in Melbourne Australia but now resident in Scottsdale Arizona.

Editor

Oh, I'm also a Kentucky Colonel. http://www.kycolonels.org

Click here to view Rob Manderson's online profile.

Discussions and Feedback

19 comments have been posted for this article. Visit http://www.codeproject.com/cpp/cfilespec.asp to post and view comments on this article.

All Topics, MFC / C++ >> C++ / MFC >> General
Updated: 27 Dec 2003