Excursions in composition: Sequential stream concatenation

devblogs.microsoft.com/oldnewthing/20070322-00

March 22, 2007

Raymond Chen

As we've seen a few times already (when building context menus and exploring fiber-based enumeration), composition is an important concept in object-oriented programming. Today, we're going to compose two sequential streams by concatenation.

There really isn't much to it. The idea is to take two streams and start by reading from the first one. When that runs out, switch to reading from the second one. Most of this is just typing. (As usual, I am using plain C++; in real life, you can save yourselves a lot of typing by using a class library of your choosing.)

We'll start with a base class that does all the boring typing related to implementing a read-only sequential stream.

```
class CROSequentialStreamBase : public ISequentialStream
{
public:
 // *** IUnknown ***
 STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
 {
  if (riid == IID_IUnknown || riid == IID_ISequentialStream) {
    *ppv = static_cast<IUnknown*>(this);
    AddRef();
    return S_OK;
  }
  *ppv = NULL;
  return E_NOINTERFACE;
 }
 STDMETHODIMP_(ULONG) AddRef()
 {
  return InterlockedIncrement(&m_cRef);
 }
 STDMETHODIMP_(ULONG) Release()
 {
  LONG cRef = InterlockedDecrement(&m_cRef);
  if (cRef == 0) delete this;
  return cRef;
 }
 // *** ISequentialStream ***
 STDMETHODIMP Write(const void *pv, ULONG cb, ULONG *pcbWritten)
 {
  if (pcbWritten) *pcbWritten = 0;
  return STG_E_ACCESSDENIED;
 }
protected:
 CROSequentialStreamBase() : m_cRef(1) { }
 virtual ~CROSequentialStreamBase() { }
 LONG m_cRef;
};
```

There's nothing exciting here at all. In addition to the boring `IUnknown` methods, we also implement `ISequentialStream::Write` by saying, "Sorry, you can't write to a read-only stream." The `ISequentialStream::Read` method is left unimplemented.

We can now cook up our `CConcatStream` class:

```
class CConcatStream : public CROSequentialStreamBase
{
public:
 CConcatStream(ISequentialStream *pstm1,
               ISequentialStream *pstm2);
 // *** ISequentialStream ***
 STDMETHODIMP Read(void *pv, ULONG cb, ULONG *pcbRead);
protected:
 ~CConcatStream();
 bool m_fFirst;
 ISequentialStream *m_pstm1;
 ISequentialStream *m_pstm2;
};
CConcatStream::CConcatStream(ISequentialStream *pstm1,
                             ISequentialStream *pstm2)
 : m_pstm1(pstm1), m_pstm2(pstm2), m_fFirst(true)
{
 assert(pstm1 != pstm2);
 m_pstm1->AddRef();
 m_pstm2->AddRef();
}
CConcatStream::~CConcatStream()
{
 m_pstm1->Release();
 m_pstm2->Release();
}
```

Our `CConcatStream` takes two sequential streams and saves them in member variables `m_pstm1` and `m_pstm2`. The real work happens in `ISequentialStream::Read` method:

```
HRESULT CConcatStream::Read(void *pv, ULONG cb, ULONG *pcbRead)
{
 ULONG cbRead;
 HRESULT hr;
 if (m_fFirst) {
  hr = m_pstm1->Read(pv, cb, &cbRead);
 } else {
  hr = m_pstm2->Read(pv, cb, &cbRead);
 }
 if ((FAILED(hr) || cbRead == 0) && m_fFirst) {
  m_fFirst = false;
  hr = m_pstm2->Read(pv, cb, &cbRead);
 }
 if (pcbRead) *pcbRead = cbRead;
 return hr;
}
```

If we are still reading the first stream, then read from the first stream. Otherwise, read from the second stream. If the first stream reaches the end, then switch to the second stream. (Checking whether the end of the stream has been reached is very annoying since

`ISequentialStream` implementations are inconsistent in the way they report the condition. Some return `S_FALSE` on a partial read; others return `S_OK`; still others return an error code. We need to check for all of these possibilities.)

And that's all there is. If you give this object two sequential streams, it will compose those two streams and act like one giant stream that is the concatenation of the two.

Let's illustrate with a simple program:

```
#include <stdio.h>
#include <windows.h>
#include <ole2.h>
#include <assert.h>
#include <shlwapi.h>
#include <tchar.h>
... insert CConcatStream class here ...
void PrintStream(ISequentialStream *pstm)
{
 ULONG cb;
 BYTE buf[256];
 while (SUCCEEDED(pstm->Read(buf, 255, &cb)) && cb) {
  buf[cb] = 0;
  printf("%s", buf);
 }
}
int __cdecl _tmain(int argc, TCHAR **argv)
{
 if(SUCCEEDED(CoInitialize(NULL)) {
  IStream *pstm1;
  if (SUCCEEDED(SHCreateStreamOnFile(argv[1], STGM_READ, &pstm1))) {
   IStream *pstm2;
   if (SUCCEEDED(SHCreateStreamOnFile(argv[2], STGM_READ, &pstm2))) {
    CConcatStream *pstm = new CConcatStream(pstm1, pstm2);
    if (pstm) {
     PrintStream(pstm);
     pstm->Release();
    }
    pstm2->Release();
   }
   pstm1->Release();
  }
  CoUninitialize();
 }
 return 0;
}
```

This program takes two file names on the command line and creates a stream for each one, then creates a `CConcatStream` out of them both, resulting in a composite stream that produces the contents of the first file, followed by the contents of the second file. When you run this program, you should see both files printed to the screen, one after the other.

Okay, there really wasn't much going on here, but we'll use this as groundwork for next time.

Exercise: What is the significance of the assertion in the constructor?

Raymond Chen

Follow