Using fibers to simplify enumerators, part 1: When life is easier for the enumerator

devblogs.microsoft.com/oldnewthing/20041229-00

December 29, 2004

Raymond Chen

The COM model for enumeration (enumeration objects) is biased towards making life easy for the consumer and hard for the producer. The enumeration object (producer) needs to be structured as a state machine, which can be quite onerous for complicated enumerators, for example, tree walking or composite enumeration.

On the other hand, the callback model for producer (used by most Win32 functions) is biased towards making life easy for the enumerator and hard for the consumer. This time, it is the consumer that needs to be structured as a state machine, which is more work if the consumer is doing something complicated with each callback. (And even if not, you have to create a context structure to pass state from the caller, through the enumerator, to the callback.)

For example, suppose we want to write a routine that walks a directory structure, allowing the caller to specify what to do at each decision point. Let's design this first using the callback paradigm:

```
#include <windows.h>
#include <shlwapi.h>
#include <stdio.h>


enum FERESULT {
 FER_CONTINUE,      // continue enumerating
                    // (if directory: recurse into it)
 FER_SKIP,          // skip this file/directory
 FER_STOP,          // stop enumerating
};


enum FEOPERATION {
 FEO_FILE,          // found a file
 FEO_DIR,           // found a directory
 FEO_LEAVEDIR,      // leaving a directory
};


typedef FERESULT (CALLBACK *FILEENUMCALLBACK)
    (FEOPERATION feo,
     LPCTSTR pszDir, LPCTSTR pszPath,
     const WIN32_FIND_DATA* pwfd,
     void *pvContext);


FERESULT EnumDirectoryTree(LPCTSTR pszDir,
    FILEENUMCALLBACK pfnCB, void* pvContext);
```

The design here is that the caller calls `EnumDirectoryTree` and provides a callback function that is informed of each file found and can decide how the enumeration should proceed.

Designing this as a callback makes life much simpler for the implementation of `EnumDirectoryTree`.

```
FERESULT EnumDirectoryTree(
    LPCTSTR pszDir,
    FILEENUMCALLBACK pfnCB, void *pvContext)
{
 FERESULT fer = FER_CONTINUE;
 TCHAR szPath[MAX_PATH];
 if (PathCombine(szPath, pszDir, TEXT("*.*"))) {
  WIN32_FIND_DATA wfd;
  HANDLE hfind = FindFirstFile(szPath, &wfd);
  if (hfind != INVALID_HANDLE_VALUE) {
   do {
    if (lstrcmp(wfd.cFileName, TEXT(".")) != 0 &&
        lstrcmp(wfd.cFileName, TEXT("..")) != 0 &&
        PathCombine(szPath, pszDir, wfd.cFileName)) {
     FEOPERATION feo = (wfd.dwFileAttributes &
                    FILE_ATTRIBUTE_DIRECTORY) ?
                    FEO_DIR : FEO_FILE;
     fer = pfnCB(feo, pszDir, szPath, &wfd, pvContext);
     if (fer == FER_CONTINUE) {
      if (feo == FEO_DIR) {
       fer = EnumDirectoryTree(szPath, pfnCB, pvContext);
       if (fer == FER_CONTINUE) {
        fer = pfnCB(FEO_LEAVEDIR, pszDir, szPath,
                    &wfd, pvContext);
       }
      }
     } else if (fer == FER_SKIP) {
      fer = FER_CONTINUE;
     }
    }
   } while (FindNextFile(hfind, &wfd));
   FindClose(hfind);
  }
 }
 return fer;
}
```

Note: I made no attempt to make this function at all efficient since that's not my point here. It's highly wasteful of stack space (which can cause problems when walking deep directory trees). This function also doesn't like paths deeper than `MAX_PATH`; fixing this is beyond the scope of this series. Nor do I worry about reparse points, which can induce infinite loops if you're not careful.

Well, that wasn't so hard to write. But that's because we made life hard for the consumer. The consumer needs to maintain state across each callback. For example, suppose you wanted to build a list of directories and their sizes (both including and excluding subdirectories).

```
class EnumState {
public:
 EnumState()
   : m_pdirCur(new Directory(NULL)) { }
 ~EnumState() { Dispose(); }
 FERESULT Callback(FEOPERATION feo,
    LPCTSTR pszDir, LPCTSTR pszPath,
    const WIN32_FIND_DATA* pwfd);
 void FinishDir(LPCTSTR pszDir);


private:


 struct Directory {
  Directory(Directory* pdirParent)
   : m_pdirParent(pdirParent)
   , m_ullSizeSelf(0)
   , m_ullSizeAll(0) { }
  Directory* m_pdirParent;
  ULONGLONG m_ullSizeSelf;
  ULONGLONG m_ullSizeAll;
 };
 Directory* Push();
 void Pop();
 void Dispose();


 Directory* m_pdirCur;
};


EnumState::Directory* EnumState::Push()
{
 Directory* pdir = new Directory(m_pdirCur);
 if (pdir) {
  m_pdirCur = pdir;
 }
 return pdir;
}


void EnumState::Pop()
{
 Directory* pdir = m_pdirCur->m_pdirParent;
 delete m_pdirCur;
 m_pdirCur = pdir;
}


void EnumState::Dispose()
{
 while (m_pdirCur) {
```

```
  Pop();
 }
}


void EnumState::FinishDir(LPCTSTR pszDir)
{
  m_pdirCur->m_ullSizeAll +=
    m_pdirCur->m_ullSizeSelf;
  printf("Size of %s is %I64d (%I64d)\n",
   pszDir, m_pdirCur->m_ullSizeSelf,
   m_pdirCur->m_ullSizeAll);
}


ULONGLONG FileSize(const WIN32_FIND_DATA *pwfd)
{
  return
    ((ULONGLONG)pwfd->nFileSizeHigh << 32) +
    pwfd->nFileSizeLow;
}


FERESULT EnumState::Callback(FEOPERATION feo,
    LPCTSTR pszDir, LPCTSTR pszPath,
    const WIN32_FIND_DATA* pwfd)
{
 if (!m_pdirCur) return FER_STOP;


 switch (feo) {
 case FEO_FILE:
  m_pdirCur->m_ullSizeSelf += FileSize(pwfd);
  return FER_CONTINUE;


 case FEO_DIR:
  if (Push()) {
   return FER_CONTINUE;
  } else {
   return FER_SKIP;
  }


 case FEO_LEAVEDIR:
  FinishDir(pszPath);


 /* Propagate size into parent */
  m_pdirCur->m_pdirParent->m_ullSizeAll +=
    m_pdirCur->m_ullSizeAll;
  Pop();
  return FER_CONTINUE;
```

```
 default:
  return FER_CONTINUE;
 }
 /* notreached */
}


FERESULT CALLBACK EnumState_Callback(
    FEOPERATION feo,
    LPCTSTR pszDir, LPCTSTR pszPath,
    const WIN32_FIND_DATA* pwfd,
    void* pvContext)
{
 EnumState* pstate =
    reinterpret_cast<EnumState*>(pvContext);
 return pstate->Callback(feo, pszDir,
            pszPath, pwfd);
}


int __cdecl main(int argc, char **argv)
{
 EnumState state;
 if (EnumDirectoryTree(TEXT("."),
        EnumState_Callback,
        &state) == FER_CONTINUE) {
  state.FinishDir(TEXT("."));
 }
 return 0;
}
```

Boy that sure was an awful lot of typing, and what's worse, the whole structure of the program has been obscured by the explicit state management. It sure is hard to tell at a glance what this chunk of code is trying to do. Instead, you have to stare at the `EnumState` class and reverse-engineer what's going on.

(Yes, I could have simplified this code a little by using a built-in stack class, but as I have already noted in the context of smart pointers, I try to present these articles in "pure" C++ so people won't get into arguments about which class library is best.)

Tomorrow, we'll look at how the world would be if the function `EnumDirectoryTree` were spec'd out by the caller rather than the enumerator!

Raymond Chen

Follow