How can I get information about the items in the Recycle Bin?

devblogs.microsoft.com/oldnewthing/20110830-00

August 30, 2011

Raymond Chen

For some reason, a lot of people are interested in programmatic access to the contents of the Recycle Bin. They never explain why they care, so it's possible that they are looking at their problem the wrong way.

For example, one reason for asking, "How do I purge an item from the Recycle Bin given a path?" is that some operation in their program results in the files going into the Recycle Bin and they want them to be deleted entirely. The correct solution is to clear the `FOF_ALLOW-UNDO` flag when deleting the items in the first place. Moving to the Recycle Bin and then purging is the wrong solution because your search-and-destroy mission may purge more items than just the ones your program put there.

The Recycle Bin is somewhat strange in that it can have multiple items with the same name. Create a text file called `TEST.TXT` on your desktop, then delete it into the Recycle Bin. Create another text file called `TEST.TXT` on your desktop, then delete it into the Recycle Bin. Now open your Recycle Bin. Hey look, you have two `TEST.TXT` files with the same path!

Now look at that original problem: Suppose the program, as part of some operation, moves the file `TEST.TXT` from the desktop to the Recycle Bin, and then the second half of the program goes into the Recycle Bin, finds `TEST.TXT` and purges it. Well, there are actually three copies of `TEST.TXT` in the Recycle Bin, and only one of them is the one you wanted to purge.

Okay, I got kind of sidetracked there. Back to the issue of getting information about the items in the Recycle Bin.

The Recycle Bin is a shell folder, and the way to enumerate the contents of a shell folder is to bind to it and enumerate its contents. The low-level interface to the shell namespace is via `IShellFolder`. There is an easier-to-use medium-level interface based on `IShellItem`, and there's a high-level interface based on `Folder` designed for scripting.

I'll start with the low-level interface. As usual, the program starts with a bunch of header files.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <shlobj.h>
#include <shlwapi.h>
#include <propkey.h>
```

The `BindToCsidl` function binds to a folder specified by a `CSIDL`. The modern way to do this is via `KNOWNFOLDER`, but just to keep you old fogeys happy, I'm doing things the classic way since you refuse to upgrade from Windows XP. (We'll look at the modern way later.)

```
HRESULT BindToCsidl(int csidl, REFIID riid, void **ppv)
{
 HRESULT hr;
 PIDLIST_ABSOLUTE pidl;
 hr = SHGetSpecialFolderLocation(NULL, csidl, &pidl);
 if (SUCCEEDED(hr)) {
  IShellFolder *psfDesktop;
  hr = SHGetDesktopFolder(&psfDesktop);
  if (SUCCEEDED(hr)) {
   if (pidl->mkid.cb) {
    hr = psfDesktop->BindToObject(pidl, NULL, riid, ppv);
   } else {
    hr = psfDesktop->QueryInterface(riid, ppv);
   }
   psfDesktop->Release();
  }
  CoTaskMemFree(pidl);
 }
 return hr;
}
```

The subtlety here is in the test for `pidl->mkid.cb`. The `IShellFolder::BindToObject` method is for binding to child objects (or grandchildren or deeper descendants). If the object you want is the desktop itself, then you can't use `IShellFolder::BindToObject` since the desktop is not a child of itself. In fact, if the object you want is the desktop itself, then *you already have the desktop*, so we just `QueryInterface` for it. It's an annoying special case which usually lurks in your code until somebody tries something like "Save file to desktop" or changes the location of a special folder to the desktop, and then boom you trip over the fact that the desktop is not a child of itself. (See further discussion below.)

Another helper function prints the display name of a shell namespace item. There isn't much interesting here either.

```
void PrintDisplayName(IShellFolder *psf,
    PCUITEMID_CHILD pidl, SHGDNF uFlags, PCTSTR pszLabel)
{
 STRRET sr;
 HRESULT hr = psf->GetDisplayNameOf(pidl, uFlags, &sr);
 if (SUCCEEDED(hr)) {
  PTSTR pszName;
  hr = StrRetToStr(&sr, pidl, &pszName);
  if (SUCCEEDED(hr)) {
   _tprintf(TEXT("%s = %s\n"), pszLabel, pszName);
   CoTaskMemFree(pszName);
  }
 }
}
```

Our last helper function retrieves a property from the shell namespace and prints it. (Obviously, if we wanted to do something other than print it, we could coerce the type to something other than `VT_BSTR`.)

```
void PrintDetail(IShellFolder2 *psf, PCUITEMID_CHILD pidl,
    const SHCOLUMNID *pscid, PCTSTR pszLabel)
{
 VARIANT vt;
 HRESULT hr = psf->GetDetailsEx(pidl, pscid, &vt);
 if (SUCCEEDED(hr)) {
  hr = VariantChangeType(&vt, &vt, 0, VT_BSTR);
  if (SUCCEEDED(hr)) {
   _tprintf(TEXT("%s: %ws\n"), pszLabel, V_BSTR(&vt));
  }
  VariantClear(&vt);
 }
}
```

Okay, now we can get down to business. The properties we will display from each item in the Recycle Bin are the item name and path, the original location (before the item was deleted), the date the item was deleted, and the size of the item.

Getting the name and path are done with various combinations of flags to `IShell-Folder::GetDisplayNameOf`, whereas getting the other properties involve talking to the shell property system. (My colleague Ben Karas covers the shell property system on his blog.) The `SHCOLUMNID` documentation says that the displaced property set applies to items which have been moved to the Recycle Bin, so we can define those column IDs based on the values provided in `shlguid.h`:

```
const SHCOLUMNID SCID_OriginalLocation =
   { PSGUID_DISPLACED, PID_DISPLACED_FROM };
const SHCOLUMNID SCID_DateDeleted =
   { PSGUID_DISPLACED, PID_DISPLACED_DATE };
```

The other property we want is `System.Size`, which the documentation says is defined as `PKEY_Size` by the `propkey.h` header file.

Okay, let's roll!

```
int __cdecl _tmain(int argc, PTSTR *argv)
{
 HRESULT hr = CoInitialize(NULL);
 if (SUCCEEDED(hr)) {
  IShellFolder2 *psfRecycleBin;
  hr = BindToCsidl(CSIDL_BITBUCKET, IID_PPV_ARGS(&psfRecycleBin));
  if (SUCCEEDED(hr)) {
   IEnumIDList *peidl;
   hr = psfRecycleBin->EnumObjects(NULL,
     SHCONTF_FOLDERS | SHCONTF_NONFOLDERS, &peidl);
   if (hr == S_OK) {
    PITEMID_CHILD pidlItem;
    while (peidl->Next(1, &pidlItem, NULL) == S_OK) {
     _tprintf(TEXT("------------------\n"));
     PrintDisplayName(psfRecycleBin, pidlItem,
                      SHGDN_INFOLDER, TEXT("InFolder"));
     PrintDisplayName(psfRecycleBin, pidlItem,
                      SHGDN_NORMAL, TEXT("Normal"));
     PrintDisplayName(psfRecycleBin, pidlItem,
                      SHGDN_FORPARSING, TEXT("ForParsing"));
     PrintDetail(psfRecycleBin, pidlItem,
                 &SCID_OriginalLocation, TEXT("Original Location"));
     PrintDetail(psfRecycleBin, pidlItem,
                 &SCID_DateDeleted, TEXT("Date deleted"));
     PrintDetail(psfRecycleBin, pidlItem,
                 &PKEY_Size, TEXT("Size"));
     CoTaskMemFree(pidlItem);
    }
   }
   psfRecycleBin->Release();
  }
  CoUninitialize();
 }
 return 0;
}
```

The only tricky part is the test for whether the call to `IShellFolder::EnumObjects` succeeded, highlighted above. According to the rules for `IShellFolder::EnumObjects`, the method is allowed to return `S_FALSE` to indicate that there are no children, in which case it sets `peidl` to `NULL`.

If you are willing to call functions new to Windows Vista, you can simplify the `BindToCsidl` function by using the helper function `SHBindToObject`. This does the work of getting the desktop folder and handling the desktop special case.

```
HRESULT BindToCsidl(int csidl, REFIID riid, void **ppv)
{
 HRESULT hr;
 PIDLIST_ABSOLUTE pidl;
 hr = SHGetSpecialFolderLocation(NULL, csidl, &pidl);
 if (SUCCEEDED(hr)) {
  hr = SHBindToObject(NULL, pidl, NULL, riid, ppv);
  CoTaskMemFree(pidl);
 }
 return hr;
}
```

But at this point, I'm starting to steal from the topic I scheduled for next time, namely modernizing this program to take advantage of some new helper functions and interfaces. We'll continue next time.

Raymond Chen

Follow