An NTFS Parser Lib

By **cyb70289**, 30 May 2010

★★★★★ 4.94 (29 votes) Rate this:

Download source code - 21.7 KB

Download demo - 134 KB

Introduction

This is a library to help in parsing an NTFS volume, as well as file records and attributes. The readers are assumed to have deep knowledge about NTFS and C++ programming.

I will not introduce NTFS concepts here as the introduction will be either a big animal or nothing at all. Search the best document about NTFS here.

Being an OS fan, I was shameful to have very little knowledge about the file system. Every time I read an OS related book, I was at a loss in the chapter "File System". The contents were either too concise for a deep understanding, or too tedious to keep reading. So I decided to write some short codes to find out what was going on in my hard disk. I picked NTFS as it's the file system in my box, and almost everyone says it's a good design, at least not a bad one.

At first, it was quite painful as there was very little documentation available. Microsoft didn't make its so called "New Technology File System" public. Only pieces of information could be found over the web. After studying the collected documents for some days, the cloud over my head scattered gradually. After some successful testing, I thought it was okay to write a library to facilitate NTFS parsing, also to deepen my knowledge.

Windows NT tries to construct an object oriented Operating System. At the very beginning, I hesitated in choosing whether to use C++ classes or traditional C procedures to fulfill the task. As an important part of the OS, it should be efficient and compact, as well as have scalability and manageability. The OS kernel must be written in C. But I'm writing a user land library, and after studying NTFS data structures thoroughly and carefully, I decided to use C++ classes to encapsulate them.

NTFS is an advanced journaling file system which fits the needs from home PCs to data servers. I haven't implemented all of its features. The following parts are not supported yet:

1. Journaling
2. Security
3. Encryption and compression
4. Some other advanced features

Demo projects

1. ntfsundel

Its purpose is to search and recover deleted files.

It seems a hard job, but it took me less than an hour to implement it by using this library, and much time was wasted on adjusting the dialog interface. Of course, this is rather a simple test program than a

About Article

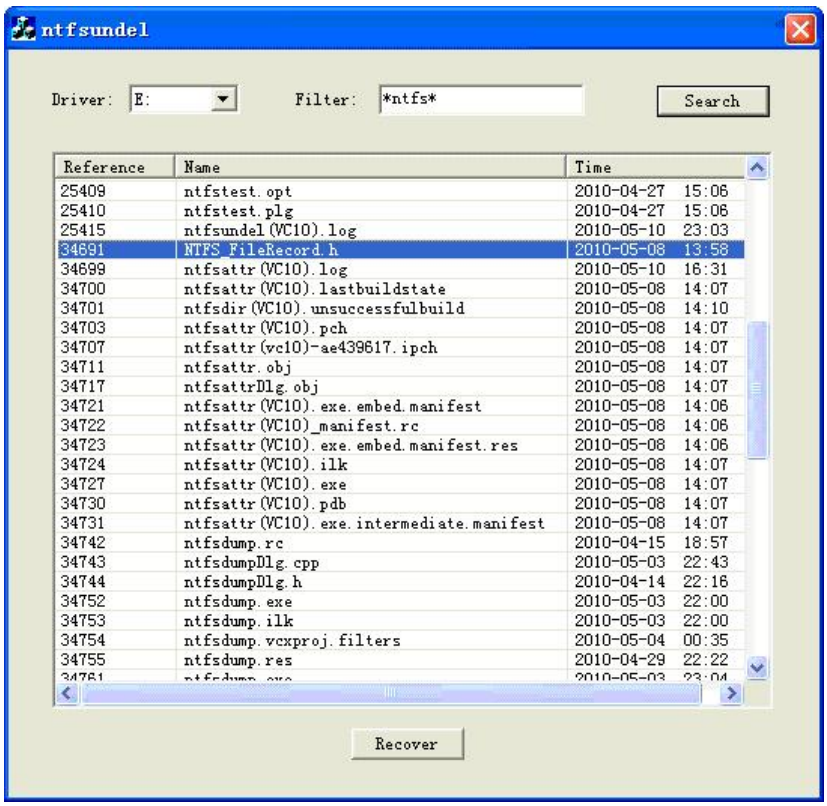
A C++ library to help in parsing an NTFS volume, file record and attributes.

Type	Article
Licence	GPL3
First Posted	15 May 2010
Views	46,538
Downloads	4,895
Bookmarked	80 times

C++ Windows Design Advanced

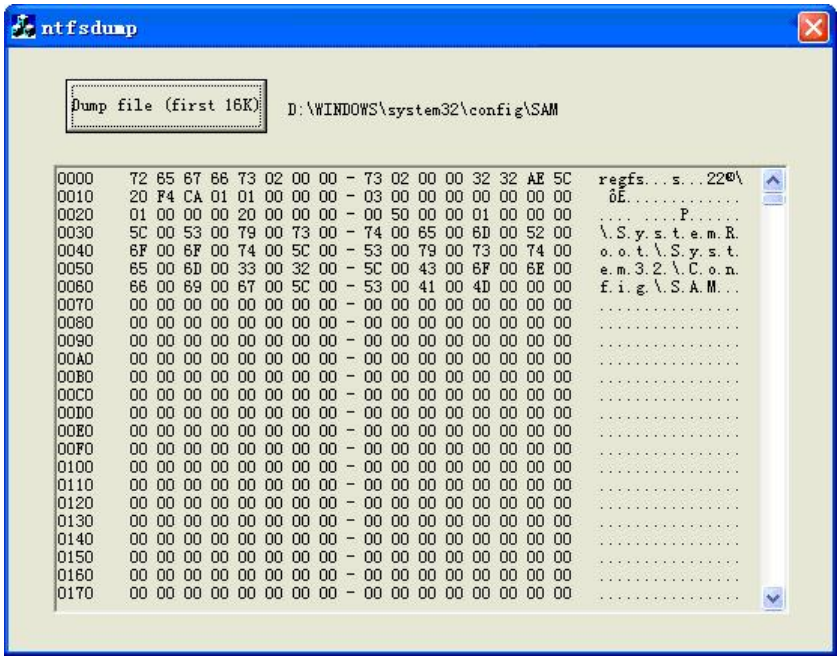
Print Email

commercial product. I didn't check if the freed clusters had been modified by another file (it's one reason why commercial tools take such a long time when analyzing a big volume).



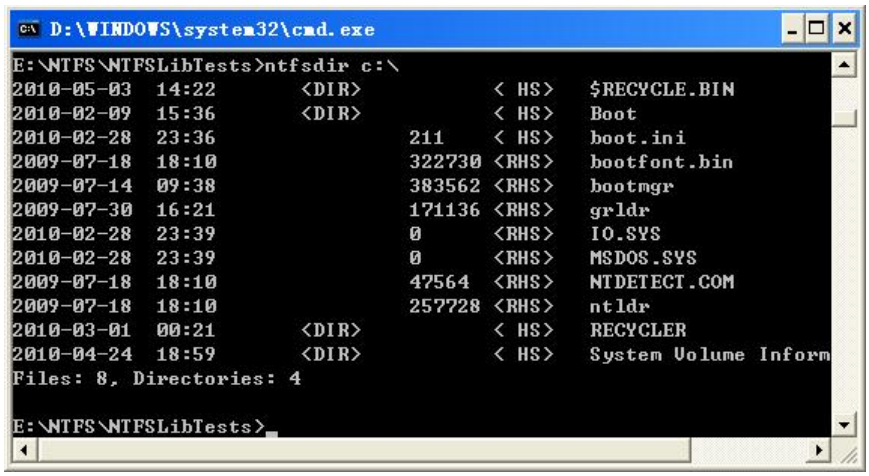
2. ntfsdump

Dump the first 16K of a file. As this library reads data directly from disk sectors, we can bypass the OS protection and peek normally inaccessible files, such as those located in "*Windows\System32\config*".



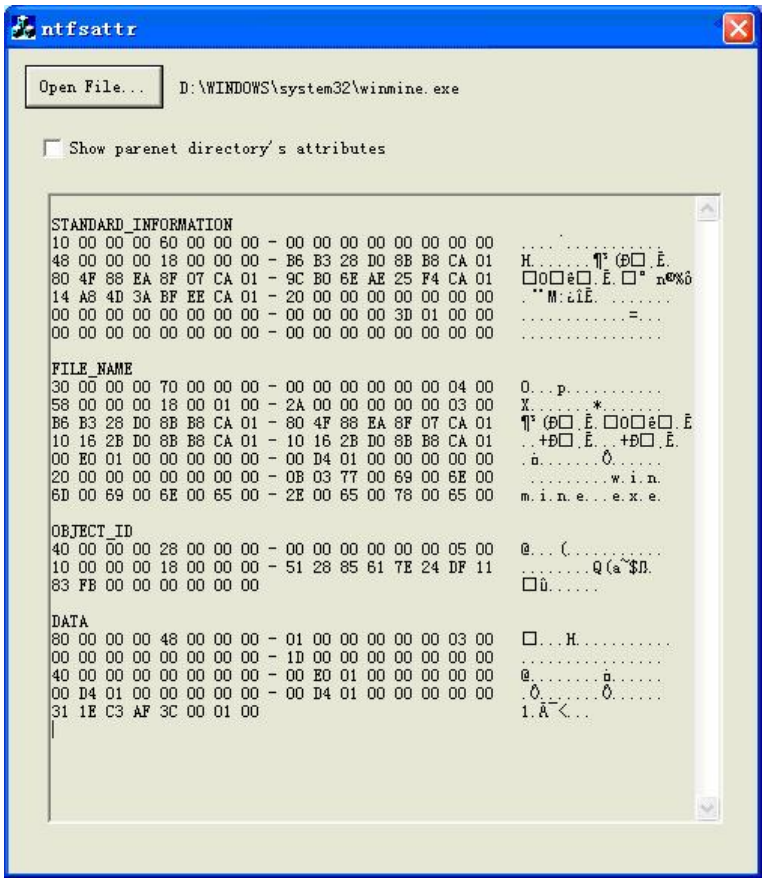
3. ntfsdir

List sub files and directories.



4. ntfsattr

List attributes of a file or a directory.



Source code

1. Source files

The source contains five *.h* files. I prefer coding directly in include files when programming C++ because it eases the deployment a lot, and looks cool too. Just include the *.h* file and everything is done, without the need to add *.cpp* files to the project. The library is part of your own source, and an unreferenced library source code is silently discarded by the compiler. Of course, it will be difficult to implement a large system this way, when classes reference each other. I don't know how Microsoft ATL achieves this goal.

1. NTFS.h

Include this file in your source. No other includes are needed.

2. NTFS_DataType.h

NTFS common data structures and data type definitions. No classes, only structures.

3. NTFS_Common.h

NTFS data structures and data type definitions specific to this library. And a single list implementation `CSList` to help in managing objects of the same type.

4. NTFS_FileRecord.h

NTFS volume and file record classes definition and implementation.

5. NTFS_Attribute.h

NTFS attributes classes and helper classes definition and implementation.

2. Coding

Having been an embedded system designer for about ten years, I am accustomed to limited system resources and digging the full capacity of hardware (think about implementing an IP stack on an 8 bit CPU running at 2MIPS with only 256 bytes of RAM). On a PC nowadays, RAM and CPU speed are not problems anymore, but I still keep the habit of writing compact code which runs as efficient and fast as possible.

To achieve this goal, many data buffers are shared between different objects in this library. To fulfill the different tasks, playing tricks with a pointer is a must, though dangerous. C++ helps us in memory management by introducing a constructor and a destructor, as well as a copy constructor, but that's not enough. Otherwise, there won't be the so called "Smart Pointer" which is just a C++ style trick about a pointer (of course, if you are not "smart" enough, it will lead to "smart" errors that are hard to discover).

I am trying to make this library more useful than a simple test. The source code and demo projects are developed in VC6.0 SP6, and can also be compiled in VC10.0. The binaries are tested in Windows XP SP3 and Windows7. I have put many tracing messages which will be shown in the Output window of Visual Studio to help debugging. The library is Unicode compatible, and can be compiled into ANSI or Unicode binaries. Define `_UNICODE` to make a Unicode build. Just like an NT kernel, NTFS uses Unicode to store file names. So a Unicode build will run faster than an ANSI one. All passed or returned pointers and references which should not be modified by the target are decorated as "`const`". The compiler will warn us if we try to modify these buffers or objects (but I offend my own rule time and time by typecasting them to non-constant pointers). And I have added validation code to prevent bad parameters and incorrect data. You cannot be too careful when handling disk volumes.

This library reads disk sectors frequently. So I will maintain some buffers to fasten data access. Though the OS has already helped us with the disk cache, a user land buffer will be a plus.

As it directly accesses the disk sectors, you must have administrator privileges to run the demo projects. In Windows7, only getting administrator privilege is not enough; an elevated privilege is required. You should be the user "Administrator" or get the elevated privilege to successfully open a volume. This library accesses the disk in read-only mode; it **should** be safe and will not harm your disk volume. Use it at your own risk.

NTFS volume and file record classes

1. CNTFSVolume

This class encapsulates a single NTFS volume.

1. `CNTFSVolume(_TCHAR volume)`

   `volume` is the volume name;, e.g.: 'C', 'D'. This is the only constructor. It does the following:

   a. Opens the volume in read-only mode, and gets a handle to directly access the disk's physical sectors.
   b. Reads BPB, does some verification, and stores the needed information.

   c. Parses NTFS metafile $Volume, reads and verifies the NTFS version.
   d. Parses the NTFS metafile $MFT, gets its $DATA attribute to locate other file records in a fragmented $MFT. NTFS tries to keep the file records continuous by reserving some buffer after $MFT. But in my eight years old Notebook, $MFT is fragmented into three parts in the system volume.

2. `BOOL IsVolumeOK() const`

   User should call this function immediately after the constructor to verify everything is OK. If this function returns `FALSE`, no other processing should be done.

3. `ULONGLONG GetRecordsCount() const`

   Returns the count of file records in this volume. It's not the sum of all the current files and directories, as deleted files may still occupy record slots.

4. `DWORD GetSectorSize() const`

   Size of disk's physical sector in bytes. Normally 512. Get from BPB.

5. `DWORD GetFileRecordSize() const`

   Size of a single file record in bytes. Normally 1024. Get from BPB.

6. `DWORD GetIndexBlockSize() const`

   Size of an index block in bytes. Normally 4096. Get from BPB.

7. `ULONGLONG GetMFTAddr() const`

   Relative start address of the $MFT metafile. Get from BPB.

8. `BOOL InstallAttrRawCB(DWORD attrType, ATTR_RAW_CALLBACK cb)`

   - `attrType`: Attribute type.
   - `cb`: Callback function.

   Return value: `TRUE` on success. `FALSE` when `attrType` is not a valid attribute type.

   Installs a volume scope callback function to be called once a specific attribute is found. Can be used to peek the raw attribute stream before it's being processed.

9. `void ClearAttrRawCB()`

   Removes all volume scope callback functions.

2. CFileRecord

Parses a single file record. It's the most important class. NTFS treats almost everything as files, even the boot sector.

1. `CFileRecord(const CNTFSVolume *volume)`

   `volume` represents which volume this file record belongs to.

2. `BOOL ParseFileRecord(ULONGLONG fileRef)`

   `fileRef` is the dile reference of the file to be parsed.

   Return value: `TRUE` on success. Otherwise `FALSE`. When this function fails, no further processing should be done.

   This function reads the file record from the disk, then verifies and patches the update sequence numbers. The user can parse as many files as possible one by one. The previously parsed data will be freed.

3. `BOOL ParseAttrs()`

   Parse selected attributes (chosen by the `SetAttrMask()` routine) of a file record. It is the biggest and most time consuming routine in the lib. All selected attributes are parsed into the corresponding C++ objects and inserted into a separate list by their type.

4. `BOOL InstallAttrRawCB(DWORD attrType, ATTR_RAW_CALLBACK cb)`

   - `attrType`: Attribute type.
   - `cb`: Callback function.

Return value: `TRUE` on success. `FALSE` when `attrType` is not valid.

Installs a file record scope callback function to be called once a specific attribute is found. Can be used to peek the raw attribute stream before it's being processed.

When `ParseAttrs()` finds an attribute, it will first lookup in `CFileRecord` to find the installed callback function and calls it. If nothing is found, it will continue searching the callback functions installed in the `CNTFSVolume` object this file record belongs to.

5. `void ClearAttrRawCB()`

   Removes all file record scope callback functions.

6. `void SetAttrMask(DWORD mask)`

   `mask` has the attributes to parse. Defined in *NTFS_Common.h* as `MASK_???`.

   User can pick the attributes to parse and discard the unwanted ones to save time and RAM. For example, you needn't waste time parsing the `$DATA` attribute if you only want to get the file's size and timestamp. `$STANDARD_INFORMATION` and `$ATTRIBUTE_LIST` will always be parsed whether they are picked or not, but unwanted attributes in `$ATTRIBUTE_LIST` will be discarded.

   This function should be called before `ParseAttrs()`.

7. `void TraverseAttrs(ATTRS_CALLBACK attrCallBack, void *context)`

   - `attrCallBack`: User defined callback function
   - `context`: context to pass to the callback function

   This routine traverses all the parsed attributes of a file record and synchronously calls the user defined callback function, and provides user the parsed C++ object of the attribute.

   This routine should be called after `ParseAttrs()`.

8. `const CAttrBase* FindFirstAttr(DWORD attrType) const`

   Find the first attribute with type "`attrType`" contained in this file record. If no attribute of "`attrType`" is found, `NULL` is returned. Once called, the internal index moves to the first element.

   This routine should be called after `ParseAttrs()`.

9. `const CAttrBase* FindNextAttr(DWORD attrType) const`

   Find the next attribute with type "`attrType`" contained in this file record. If no more attribute of "`attrType`" is found, `NULL` is returned. Once called, the internal index is moved to next.

   This routine should be called after `FindFirstAttr()`.

   Collapse | Copy Code

```
CAttrBase *ab = FindFirstAttr(ATTR_TYPE_FILENAME)
while (ab)
{
    // process ab here
    ab = FindNextAttr(ATTR_TYPE_FILENAME);
}
```

   The MFC `CFileFind` class is really a bad design and error prone. So I didn't follow its style.

10. `int GetFileName(_TCHAR *buf, DWORD bufLen) const`

    - `buf`: Name buffer to hold the returned file name.
    - `bufLen`: Name buffer size in characters (not bytes!)

    Return value:

    - > 0: Name length in characters.
    - = 0: This file is unnamed.
    - < 0: Buffer size is less than the file name size, the negative value is the wanted buffer size. For example, a return value of -20 means you need a buffer with its size at least 20 characters.

    A single file record may have several file names (`$FILE_NAME` attribute). The first Win32 name will be returned.

11. `ULONGLONG GetFileSize() const`

    Get the file size in bytes. Get from the `$FILE_NAME` attribute.

12. `void GetFileTime(FILETIME *writeTm, FILETIME *createTm = NULL, FILETIME *accessTm = NULL) const`

    Get file last alteration time, creation time, and last access time. The time is already converted to the time zone set in the system. Get from the `$STANDARD_INFORMATION` attribute.

13. `void TraverseSubEntries(SUBENTRY_CALLBACK seCallBack) const`

    Traverse all the subentries located in a file record (a directory file) and synchronously call the user defined callback function, and provide user all the subentries encapsulated by the `CIndexEntry` class. Useful in enumerating sub files and directories. `$INDEX_ROOT` and `$INEX_ALLOCATION` attributes must have been parsed already (see `SetAttrMask()`).

14. `const BOOL FindSubEntry(const _TCHAR *fileName, CIndexEntry &ieFound) const`

    - `fileName`: Sub file name to find
    - `ieFound`: `CIndexEntry` object found

    Return value: `TRUE` when found, otherwise `FALSE`.

    It is used to find a sub file or directory. `$INDEX_ROOT` and `$INEX_ALLOCATION` attributes must have been parsed already (see `SetAttrMask()`).

15. `const CAttrBase* FindStream(_TCHAR *name = NULL)`

    `name` is the file data stream name. `NULL` for unnamed stream.

    Find the specific data stream by name. NTFS files may have several data streams (`$DATA` attribute). File content is always located in an unnamed stream. The `$DATA` attribute must have been parsed already (see `SetAttrMask()`).

16. `BOOL IsDeleted() const`

    Check if this file record is deleted.

17. `BOOL IsDirectory() const`

    Check if this file record is a directory.

18. `BOOL IsReadOnly() const`

    Check if it's a read-only file. Get from the `$STANDARD_INFORMATION` attribute.

19. `BOOL IsHidden() const`

    Check if it's a hidden file. Get from the `$STANDARD_INFORMATION` attribute.

20. `BOOL IsSystem() const`

    Check if it's a system file. Get from the `$STANDARD_INFORMATION` attribute.

21. `BOOL IsCompressed() const`

    Check if it's a compressed file. Get from the `$STANDARD_INFORMATION` attribute.

22. `BOOL IsEncrypted() const`

    Check if it's an encrypted file. Get from the `$STANDARD_INFORMATION` attribute.

23. `BOOL IsSparse() const`

    Check if it's a sparse file. Get from the `$STANDARD_INFORMATION` attribute.

NTFS attributes classes

Collapse | Copy Code

```
Attributes        Class

$STANDARD_INFORMATION    CAttr_StdInfo
$ATTRIBUTE_LIST        CAttr_AttrList<TYPE_RESIENT>
$FILE_NAME        CAttr_FileName
$VOLUME_NAME        CAttr_VolName
$VOLUME_INFORMATION    CAttr_VolInfo
$DATA            CAttr_Data<TYPE_RESIDENT>
$INDEX_ROOT        CAttr_IndexRoot
$INDEX_ALLOCATION        CAttr_IndexAlloc
$BITMAP            CAttr_Bitmap<TYPE_RESIENT>
```

NTFS attributes are classified into resident (`CAttrResident`) and nonresident (`CAttrNonResident`). Resident and nonresident attributes share a common header (`CAttrBase`). All attribute classes are derived from `CAttrResident` or `CAttrNonResident`, which are derived from `CAttrBase`. Some attributes, such as `$DATA` and `$ATTRIBUTE_LIST` can be resident or nonresident; these classes use a template parameter as their base class.

1. CAttrBase

Base class of all the attribute classes.

1. `CAttrBase(const ATTR_HEADER_COMMON *ahc, const CFileRecord *fr)`
   - `ahc`: Points to the attribute header buffer.
   - `fr`: The file record which owns this attribute.
2. `virtual __inline ULONGLONG GetDataSize(ULONGLONG *allocSize = NULL) const = 0`

   `allocSize` is the allocated size of the data in bytes. Just leave this parameter blank if you don't want it.

   Return value: Actual size of the data in bytes.

   Get size of this attribute's data in bytes. It's declared as a pure virtual function. The derived classes `CAttrResident` and `CAttrNonResident` will actually implement this function. Thanks to polymorphism introduced by C++, with this function and the following function `ReadData()`, resident and non-resident attributes can access their data in the same interface, though they divert so much.
3. `virtual BOOL ReadData(const ULONGLONG &offset, void *bufv, DWORD bufLen, DWORD *actural) const = 0`
   - `offset`: Start address of the read pointer relative to beginning in bytes.
   - `bufv`: User provided buffer to receive the data.
   - `bufLen`: User provided buffer size in bytes.
   - `actural`: The actual size of data read. Sorry for the misspelling. I got it right now when Microsoft Word tells me, but I'm too lazy to find and replace all the errors in my source code. I suggest Microsoft add spell checking in Visual Studio to help us non-English speaking guys, he he.

   Return value: `TRUE` on success, otherwise `FALSE`.

   Read attribute data into a buffer.
4. Other exported routines:

   Collapse | Copy Code

```
__inline const ATTR_HEADER_COMMON* GetAttrHeader() const
__inline DWORD GetAttrType() const
__inline DWORD GetAttrTotalSize() const
__inline BOOL IsNonResident() const
__inline WORD GetAttrFlags() const
int GetAttrName(char *buf, DWORD bufLen) const
int GetAttrName(wchar_t *buf, DWORD bufLen) const
```

   Get attribute name. The return value obeys the same rule as `CFileRecord::GetFileName()`

   Collapse | Copy Code

```
__inline BOOL IsUnNamed() const
```

   Check if this attribute is unnamed.

2. CAttrResident

Base class of all resident attribute classes.

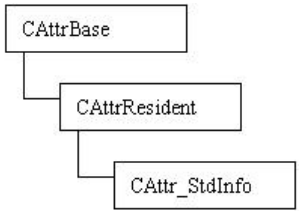
Implements the virtual functions `GetDataSize()` and `ReadData()` specific to resident attributes.

3. CAttrNonResident

Base class of all non-resident attribute classes. Implements the virtual functions `GetDataSize()` and `ReadData()` specific to non-resident attributes. It's much more complicated than `CAttrResident`'s implementation, as it should parse data runs and build a list to hold the information. I don't think the

NTFS data run is a good design, because the saved disk space cannot compensate for the wasted parsing time.

4. CAttr_StdInfo

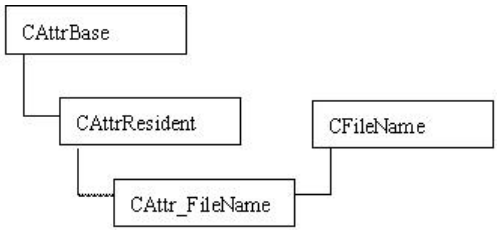


Implements the `$STANDARD_INFORMATION` attribute. Derived from `CAttrResident`. Exported functions:

Collapse | Copy Code

```
void GetFileTime(FILETIME *writeTm,
     FILETIME *createTm = NULL, FILETIME *accessTm = NULL) const
__inline DWORD GetFilePermission() const
__inline BOOL IsReadOnly() const
__inline BOOL IsHidden() const
__inline BOOL IsSystem() const
__inline BOOL IsCompressed() const
__inline BOOL IsEncrypted() const
__inline BOOL IsSparse() const
```

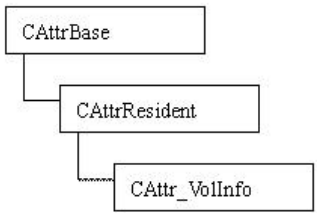
5. CAttr_FileName



Implements the `$FILE_NAME` attribute. Derived from `CAttrResident` and the `CFileName` helper class.

All useful functions are located in the `CFileName` base class which will be introduced later. File permissions and times located in a `$FILE_NAME` attribute will only be updated when the file name is changed, so related functions derived from `CFileName` are declared again as "`private`" in `CAttr_FileName` to prevent user from getting the wrong information. `$STANDARD_INFORMATION` and index entry keep the updated file permission and timestamp.

6. CAttr_VolInfo



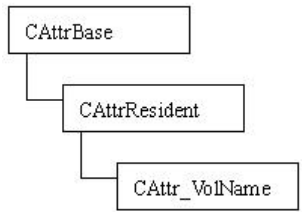
Implements the `$VOLUME_INFORMATION` attribute. Derived from `CAttrResident`. Exported functions:

Collapse | Copy Code

```
__inline WORD GetVersion()
```

Returns the NTFS volume version. High byte holds the major version, low byte the minor. In Windows XP and Windows7, the NTFS version is 3.1, Windows 2000 is 3.0, and Windows NT 1.2. NTFS volumes with version less than 3.0 is not supported by this library.

7. CAttr_VolName



Implements the `$VOLUME_NAME` attribute. Derived from `CAttrResident`.

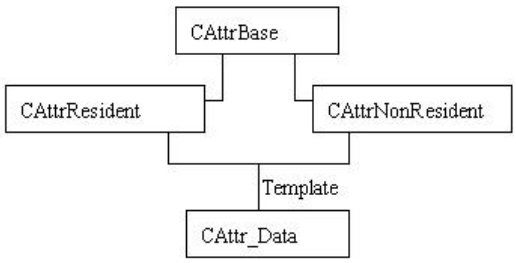
Exported functions:

Collapse | Copy Code

```
__inline int GetName(wchar_t *buf, DWORD len) const
__inline int GetName(char *buf, DWORD len) const
```

Get the Unicode or ANSI volume name. The return value obeys the same rule as `CFileRecord::GetFileName()`.

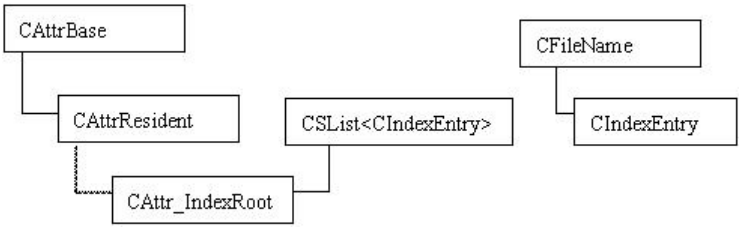
8. CAttr_Data



Implements the `$DATA` attribute. Derived from a template class which is `CAttrResident` or `CAttrNonResident`.

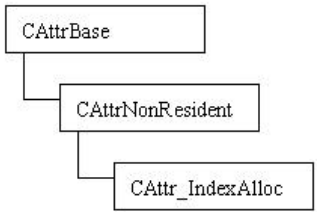
`GetDataSize()` and `ReadData()` are derived from the template base class. We only need these two functions when handling the `$DATA` attribute.

9. CAttr_IndexRoot



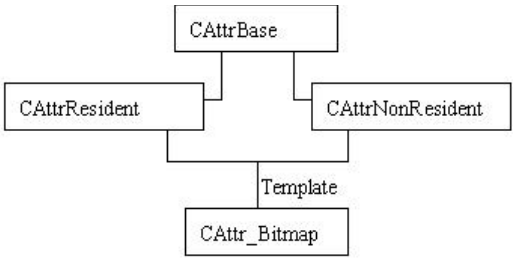
Implements the `$INDEX_ROOT` attribute. Derived from the `CAttrResident` and `CIndexEntryList` helper classes. All useful functions are located in the `CIndexEntry` object held in `CIndexEntryList` which will be introduced later.

10. CAttr_IndexAlloc



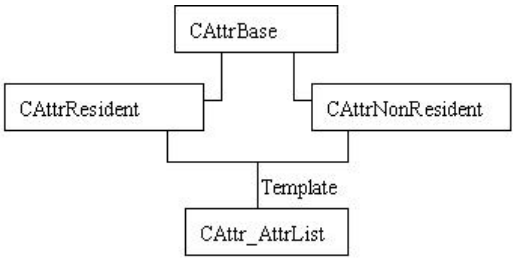
Implements the `$INDEX_ALLOCATION` attribute. Derived from `CAttrNonResident`.

11. CAttr_Bitmap



Implements the `$BITMAP` attribute. Derive from a template class which is `CAttrResident` or `CAttrNonResident`.

12. CAttr_AttrList



Implements the `$ATTRIBUTE_LIST` attribute. Derive from a template class which is `CAttrResident` or `CAttrNonResident`.

This is the most complicated attribute to process because it deals with a file record and all other attributes. But the implementation is concise, and the code is short.

User needn't care about this attribute; all parsed sub attributes will be inserted into the parent file record's attribute list, just as they are directly contained in the same file record.

Helper classes

1. CFileName

This class helps `CAttr_FileName` and `CIndexEntry` to process file name related information.

Exported functions:

Collapse | Copy Code

```
int Compare(const wchar_t *fn) const
int Compare(const char *fn) const
```

Compare the file name with the input string. Return 0 if they match, negative if the file name is smaller than the input string, and positive otherwise. This routine is used to search a specific file in the B+ tree constructed by the index root and index allocation.

Collapse | Copy Code

```
__inline ULONGLONG GetFileSize() const
__inline DWORD GetFilePermission() const
__inline BOOL IsReadOnly() const
__inline BOOL IsHidden() const
__inline BOOL IsSystem() const
__inline BOOL IsDirectory() const
__inline BOOL IsCompressed() const
__inline BOOL IsEncrypted() const
__inline BOOL IsSparse() const
int GetFileName(char *buf, DWORD bufLen) const
int GetFileName(wchar_t *buf, DWORD bufLen) const
```

Get the Unicode or ANSI file name. The return value obeys the same rule as `CFileRecord::GetFileName()`.

Collapse | Copy Code

```
__inline BOOL HasName() const
```

Check if it contains a file name or is unnamed.

Collapse | Copy Code

```
__inline BOOL IsWin32Name() const
```

File names which cannot fit into the DOS 8.3 format will have a DOS alias name. For example, the Win32 name "*C:\Program files*" will have a DOS compatible file name "*C:\Progra~1*". Use this function to check if it contains a legal Win32 name.

Collapse | Copy Code

```
void GetFileTime(FILETIME *writeTm, FILETIME *createTm = NULL,
                 FILETIME *accessTm = NULL) const
```

2. CIndexEntry

This class encapsulates a single index entry of the file name. It is derived from `CFileName`, and all `CFileName` exported functions can be used directly.

Exported functions:

Collapse | Copy Code

```
__inline ULONGLONG GetFileReference() const
```

Get the file reference of this index entry.

Collapse | Copy Code

```
__inline BOOL IsSubNodePtr() const
```

Check if the index entry points to sub nodes. These entries link different index blocks into a B+ tree.

Collapse | Copy Code

```
__inline ULONGLONG GetSubNodeVCN() const
```

Use this function to locate the sub-node index block.

3. CIndexBlock

This class helps in parsing a single index block into a list of `CIndexEntry`.

License

About the Author

cyb70289

China

From Shanghai, China

Article Top

Comments and Discussions

You must Sign In to use this message board.

Search this forum Go

Profile popups Spacing Relaxed Noise Medium Layout Normal Per page 25 Update

First Prev Next

How to work with the bitmap	Jameswilliam	22-Apr-14 1:43
great articles with severe bugs	oksmartmaster	21-Apr-14 17:14
Whether thread safety is taken into account?	Member 10251888	19-Sep-13 23:25
怎样快速扫描$MFT	marszhou	18-Jul-13 20:58
This is good for learning	reneeculver	6-Feb-13 12:12
My vote of 5	mimsdev	28-Dec-12 4:25
My vote of 5	justdownloads	19-Dec-12 8:57
How about one $MFT entry with one attribute list $20 and inside this attribute has two DATA ($80) which locate in different $MFT entry	JoHung	2-Dec-12 22:52
My vote of 5	gndnet	8-Nov-12 3:27
What about NTFS version 3.1?	PeterB78	9-Jul-12 4:54
My vote of 5	yanghuic	8-Jul-12 23:19
My vote for 5	siqiao	19-Jun-12 15:39
Set filesize to 0	viki1987	17-May-12 21:53
My vote of 5	dxFrety	6-Jul-11 14:31
2TB Virtual disk	nhchmg	27-Jun-11 20:14
Licence [modified]	hjaiuyg	7-Feb-11 22:59
Re: Licence	AGNUcius	11-Feb-11 7:01
Index allocation	Epoque	12-Jan-11 8:04
Re: Index allocation	cyb70289	14-Jan-11 0:47
how to process bitmap, thanks	fjb2080	26-Oct-10 22:00
Re: how to process bitmap, thanks	cyb70289	27-Oct-10 1:39
TraverseSubEntries bug	Sadistic-X	15-Aug-10 5:52
Re: TraverseSubEntries bug	cyb70289	15-Aug-10 18:41
Re: TraverseSubEntries bug	Sadistic-X	16-Aug-10 2:07
Re: TraverseSubEntries bug	cyb70289	16-Aug-10 16:29

Last Visit: 31-Dec-99 18:00 Last Update: 6-May-14 7:29 Refresh 1 2 3 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.