

Directory Management

What do we have for this session?

Introduction

Creating and Deleting Directories

Directory Handles

Reparse Points

Reparse Point Tags

Tag Contents

Reparse Point Operations

Reparse Points and File Operations

Using Directory Management Functions

Changing the Current Directory Program Example

Listing the Files in a Directory Program Example

Moving Directories Program Example

Obtaining Directory Change Notifications Program Example

Retrieving and Changing File Attributes Program Example

Directory Management Reference

Directory Management Control Codes

Directory Management Functions

Directory Management Macros

Directory Management Structures

Introduction

A directory is a **hierarchical collection of directories and files**. The only constraint on the number of files that can be contained in a single directory is **the physical size of the disk** on which the directory is located.

A directory that contains one or more directories is the **parent of the contained directory** or directories, and each contained directory is a **child of the parent directory**. The hierarchical structure of directories is referred to as a **directory tree**.

The NTFS file system implements the **logical link between a directory and the files it contains as a directory entry table**. When a file is **moved into a directory**, an entry is created in the table for the moved file and the name of the file is placed in the entry. When a file contained in a directory is **deleted**, the name and entry corresponding to the deleted file is also deleted from the table. **More than one entry for a single file can exist in a directory entry table**. If an additional entry is created in the table for a file, that entry is referred to as a **hard link to that file**. There is no limit to the number of hard links that can be created for a single file. Directories can also contain **junctions** and **reparse points**.

Creating and Deleting Directories

An application can programmatically create and delete directories. To create a new directory, use the CreateDirectory(), CreateDirectoryEx(), or CreateDirectoryTransacted() function. A directory is

given the name specified when it is created. The conventions for naming a directory follow the conventions for naming a file.

To delete an existing directory, use the `RemoveDirectory()` or `RemoveDirectoryTransacted()` function. Before removing a directory, you must ensure that the directory is empty and that you have the delete access privilege for the directory. To do the latter, call the `GetSecurityInfo()` function.

Directory Handles

Whenever a process creates or opens a directory object, it receives a handle to the object.

To obtain a handle to an existing directory, call the `CreateFile()` function with the `FILE_FLAG_BACKUP_SEMANTICS` flag. You can pass a directory handle to the following functions:

1. `BackupRead()` - The `BackupRead()` function can be used to back up a file or directory, including the security information. The function reads data associated with a specified file or directory into a buffer, which can then be written to the backup medium using the `WriteFile()` function.
2. `BackupSeek()` - The `BackupSeek()` function seeks forward in a data stream initially accessed by using the `BackupRead()` or `BackupWrite()` function.
3. `BackupWrite()` - The `BackupWrite` function can be used to restore a file or directory that was backed up using `BackupRead()`. Use the `ReadFile()` function to get a stream of data from the backup medium, and then use `BackupWrite()` to write the data to the specified file or directory.
4. `GetFileInformationByHandle()` - Retrieves file information for the specified file. For a more advanced version of this function, see `GetFileInformationByHandleEx()`. To set file information using a file handle, see `SetFileInformationByHandle()`.
5. `GetFileSize()` - Retrieves the size of the specified file, in bytes. It is recommended that you use `GetFileSizeEx()`.
6. `GetFileTime()` - Retrieves the date and time that a file or directory was created, last accessed, and last modified.
7. `GetFileType()` - Retrieves the file type of the specified file.
8. `ReadDirectoryChangesW()` - Retrieves information that describes the changes within the specified directory. The function does not report changes to the specified directory itself.
9. `SetFileTime()` - Sets the date and time that the specified file or directory was created, last accessed, or last modified.

Reparse Points

A file or directory can contain a **reparse point**, which is a **collection of user-defined data**. The format of this data is **understood by the application which stores the data**, and a **file system filter**, which you install to interpret the data and process the file. When an application sets a reparse point, it stores this data, plus a **reparse tag**, which uniquely identifies the data it is storing. When the file system opens a file with a reparse point, it attempts to find the file system filter associated with the data format identified by the reparse tag. If a file system filter is found, the filter processes the file as directed by the reparse data. If a file system filter is not found, the file open operation fails.

For example, **reparse points are used to implement NTFS file system links and the Microsoft Remote Storage Server (RSS)**. RSS uses an administrator-defined set of rules to move infrequently used files to long term storage, such as tape or optical media. It uses reparse points to store information about the file in the file system. This information is stored in a stub file that contains a reparse point whose data points to the device where the actual file is now located. The file system

filter can use this information to retrieve the file. **Reparse points are also used to implement mounted folders.** The following restrictions apply to reparse points:

1. Reparse points can be established for a directory, but the directory must be empty. Otherwise, the NTFS file system fails to establish the reparse point. In addition, you cannot create directories or files in a directory that contains a reparse point.
2. Reparse points and extended attributes are mutually exclusive. The NTFS file system cannot create a reparse point when the file contains extended attributes, and it cannot create extended attributes on a file that contains a reparse point.
3. Reparse point data, including the tag and optional GUID, cannot exceed 16 kilobytes. Setting a reparse point fails if the amount of data to be placed in the reparse point exceeds this limit.
4. There is a limit of 31 reparse points on any given path.

Reparse Point Tags

Each reparse point has **an identifier tag** so that you can efficiently differentiate between the different types of reparse points, without having to examine the user-defined data in the reparse point. The system uses **a set of predefined tags** and **a range of tags reserved for Microsoft**. If you use any of the reserved tags when setting a reparse point, the operation fails. Tags not included in these ranges are not reserved and are available for your application.

When you set a reparse point, you must **tag the data to be placed in the reparse point**. After the reparse point has been established, a new set operation fails if the tag for the new data does not match the tag for the existing data. If the tags match, the set operation overwrites the existing reparse point.

To retrieve the reparse point tag, use the FindFirstFile() function. If the dwFileAttributes member includes the FILE_ATTRIBUTE_REPARSE_POINT attribute, then the dwReserved0 member specifies the reparse point.

Tag Contents

Reparse tags are stored as DWORD values. The bits define certain attributes, as shown in the following diagram.



The low 16 bits determine the kind of reparse point. The high 16 bits have 12 bits reserved for future use and 4 bits that denote specific attributes of the tags and the data represented by the reparse point. The following table describes these bits.

Bit	Description
M	Microsoft bit. If this bit is set, the tag is owned by Microsoft. All other tags must use zero for this bit.
R	Reserved; must be zero for all non-Microsoft tags.
N	Name surrogate bit. If this bit is set, the file or directory represents another named entity in the system.

The following macros exist to assist in testing tags:

1. `IsReparseTagMicrosoft()` - Determines whether a reparse point tag indicates a Microsoft reparse point.
2. `IsReparseTagNameSurrogate()` - Determines whether a tag's associated reparse point is a surrogate for another named entity (for example, a mounted folder).

Each macro returns a nonzero value if the associated bit is set. The following are Microsoft's predefined reparse tag values; they are defined in `Winnt.h`:

1. `IO_REPARSE_TAG_DFS`
2. `IO_REPARSE_TAG_DFSR`
3. `IO_REPARSE_TAG_HSM`
4. `IO_REPARSE_TAG_HSM2`
5. `IO_REPARSE_TAG_MOUNT_POINT`
6. `IO_REPARSE_TAG_SIS`
7. `IO_REPARSE_TAG_SYMLINK`

To ensure uniqueness of tags, Microsoft provides a mechanism to distribute new tags as explained in the Installable File System (IFS) Kit.

Reparse Point Operations

To determine whether a **file system supports reparse points**, call the `GetVolumeInformation()` function and examine the `FILE_SUPPORTS_REPARSE_POINTS` bit flag.

The `DeviceIoControl()` function enables you to set, modify, obtain, and remove reparse points. The following table describes the reparse point operations that you can perform using `DeviceIoControl()`.

Operation	Description
<code>FSCTL_SET_REPARSE_POINT</code>	Allows the calling program to set a new reparse point, or to modify an existing one.
<code>FSCTL_GET_REPARSE_POINT</code>	Obtains the information stored in an existing reparse point.
<code>FSCTL_DELETE_REPARSE_POINT</code>	Removes an existing reparse point.

If you are modifying, getting, or deleting a reparse point, you must specify the same reparse tag in the operation that is contained in the file. Otherwise, the operation will fail with the error `ERROR_REPARSE_TAG_MISMATCH`. If you are modifying or deleting a reparse point, you must also specify the reparse GUID in the operation that is contained in the file. Otherwise, the operation will fail with the error `ERROR_REPARSE_ATTRIBUTE_CONFLICT`.

To determine whether a file or directory contains a reparse point, use the `GetFileAttributes()` function. If the file or directory has an associated reparse point, the `FILE_ATTRIBUTE_REPARSE_POINT` attribute is set.

To overwrite an existing reparse point without already having a handle to the file or directory, call `CreateFile()` with `FILE_FLAG_OPEN_REPARSE_POINT`. This flag allows you to open the file whether or not the corresponding file system filter is installed and working correctly.

Reparse Points and File Operations

Reparse points enable **file system behavior that departs from the behavior most Windows developers may be accustomed to**, therefore being aware of these behaviors when writing applications that manipulate files is vital to robust and reliable applications intended to access file systems that support reparse points. The extent of these considerations will depend on the specific implementation and associated file system filter behavior of a particular reparse point, which can be user-defined. Consider the following examples regarding NTFS **reparse point implementations**, which include **mounted folders**, **linked files**, and the **Microsoft Remote Storage Server**:

1. Backup applications that use file streams should specify `BACKUP_REPARSE_DATA` in the `WIN32_STREAM_ID` structure when backing up files with reparse points.
2. Applications that use the `CreateFile()` function should specify the `FILE_FLAG_OPEN_REPARSE_POINT` flag when opening the file if it is a reparse point.
3. The process of defragmenting files requires special handling for reparse points.
4. Virus detection applications should search for reparse points that indicate linked files.
5. Most applications should take special actions for files that have been moved to long-term storage, if only to notify the user that it may take a while to retrieve the file.
6. The `OpenFileById()` function will either open the file or the reparse point, depending on the use of the `FILE_FLAG_OPEN_REPARSE_POINT` flag.
7. Symbolic links, as reparse points, have certain programming considerations specific to them.
8. Volume management activities for reading journal records require special handling for reparse points when using the `USN_RECORD` and `READ_USN_JOURNAL_DATA` structures.

Using Directory Management APIs

The following program examples demonstrate the use of the directory management functions through program examples:

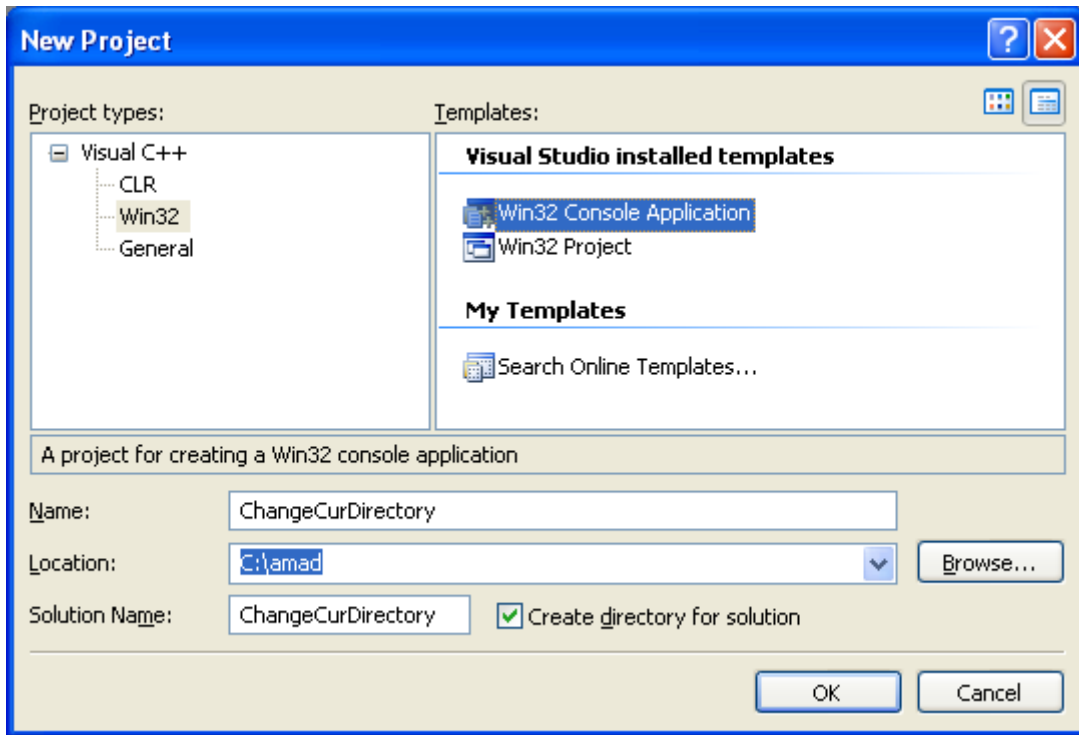
1. Changing the Current Directory
2. Listing the Files in a Directory
3. Moving Directories
4. Obtaining Directory Change Notifications
5. and more

Changing the Current Directory Program Example

The directory at the end of the active path is called the current directory; it is the directory in which the active application started, unless explicitly changed. An application can determine which directory is current by calling the `GetCurrentDirectory()` function. It is sometimes necessary to use the `GetFullPathName()` function to ensure the drive letter is included if the application requires it. Although each process can have only one current directory, if the application switches volumes using the `SetCurrentDirectory()` function, the system remembers the last current path for each volume (drive letter). This behavior will manifest itself only when specifying a drive letter without a fully qualified path when changing the current directory point of reference to a different volume. This applies to either `Get()` or `Set()` operations.

An application can change the current directory by calling the `SetCurrentDirectory()` function. The following example demonstrates the use of `GetCurrentDirectory()` and `SetCurrentDirectory()`.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.
Next, add the following source code.

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

#define BUFSIZE MAX_PATH

int wmain(int argc, WCHAR **argv)
{
    WCHAR Buffer[BUFSIZE];
    DWORD dwRet;

    if(argc != 2)
    {
        wprintf(L"Changing directory and back to the original path\n");
        wprintf(L"Usage: %s <dir>\n", argv[0]);
        wprintf(L"Example: %s D:\\test\n", argv[0]);
        return 1;
    }

    dwRet = GetCurrentDirectory(BUFSIZE, Buffer);

    if(dwRet == 0)
    {
        wprintf(L"GetCurrentDirectory failed (%d)\n", GetLastError());
        return 1;
    }
}
```

```
if(dwRet > BUFSIZE)
{
    wprintf(L"Buffer too small; need %d characters\n", dwRet);
    return 1;
}

if(!SetCurrentDirectory(argv[1]))
{
    wprintf(L"SetCurrentDirectory failed,error %d\n", GetLastError());
    return 1;
}

wprintf(L"Set current directory to %s\n", argv[1]);
// Verify
system("dir");

if(!SetCurrentDirectory(Buffer))
{
    wprintf(L"SetCurrentDirectory failed (%d)\n", GetLastError());
    return 1;
}

wprintf(L"\nBack to the previous directory,\n\n %s\n\n", Buffer);
// Verify
system("dir");

return 0;
}
```

Build and run the project. The following screenshot is a sample output.

```

C:\WINDOWS\system32\cmd.exe

C:\amad\ChangeCurDirectory\Debug>ChangeCurDirectory "C:\Documents and Settings"
Set current directory to C:\Documents and Settings
Volume in drive C has no label.
Volume Serial Number is 54C8-D37A

Directory of C:\Documents and Settings

01/12/2008  05:46 PM    <DIR>          .
01/12/2008  05:46 PM    <DIR>          ..
01/09/2008  11:22 PM    <DIR>          Administrator
01/12/2008  05:46 PM    <DIR>          All Users
11/21/2009  01:18 AM    <DIR>          mike spoon
               0 File(s)                0 bytes
               5 Dir(s)  66,500,472,832 bytes free

Back to the previous directory.

C:\amad\ChangeCurDirectory\Debug

Volume in drive C has no label.
Volume Serial Number is 54C8-D37A

Directory of C:\amad\ChangeCurDirectory\Debug

11/15/2009  10:51 PM    <DIR>          .
11/15/2009  10:51 PM    <DIR>          ..
11/21/2009  11:26 AM                29,696 ChangeCurDirectory.exe
11/21/2009  11:26 AM            327,520 ChangeCurDirectory.ilk
11/21/2009  11:26 AM            470,016 ChangeCurDirectory.pdb
               3 File(s)            827,232 bytes
               2 Dir(s)  66,500,472,832 bytes free

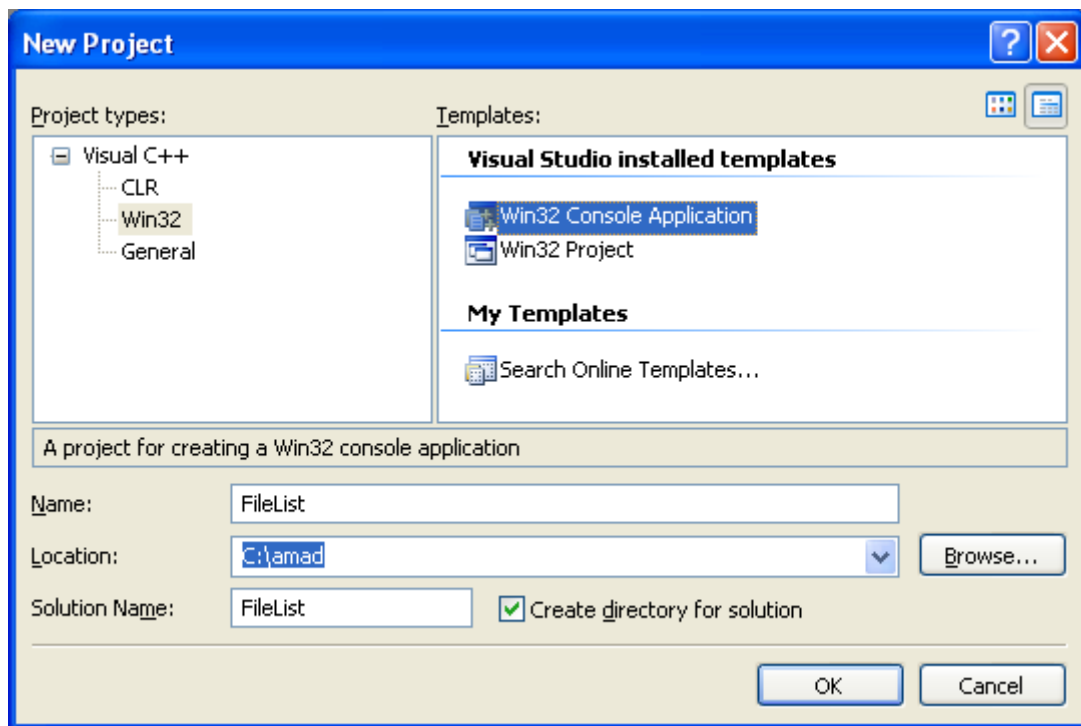
C:\amad\ChangeCurDirectory\Debug>

```

Listing the Files in a Directory Program Example

The following program example calls FindFirstFile(), FindNextFile(), and FindClose() to list files in a specified directory.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
// For StringCchLength(), StringCchCopy(), StringCchCat(),
// StringCchPrintf() - safer versions
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int wmain(int argc, WCHAR *argv[])
{
    WIN32_FIND_DATA ffd;
    LARGE_INTEGER filesize;
    WCHAR szDir[MAX_PATH];
    size_t length_of_arg;
    HANDLE hFind = INVALID_HANDLE_VALUE;
    DWORD dwError=0;

    // If the directory is not specified as a command-line argument,
    // print usage
    if(argc != 2)
    {
        wprintf(L"\nUsage: %s <directory name>\n", argv[0]);
        return (-1);
    }

    // Check that the input path plus 3 is not longer than MAX_PATH.
    // Three characters are for the "\" plus NULL appended below
    StringCchLength(argv[1], MAX_PATH, &length_of_arg);
```

```
if (length_of_arg > (MAX_PATH - 3))
{
    wprintf(L"\nDirectory path is too long.\n");
    return (-1);
}

wprintf(L"\nTarget directory is %s\n\n", argv[1]);

// Prepare string for use with FindFile functions. First, copy the
// string to a buffer, then append '*' to the directory name
StringCchCopy(szDir, MAX_PATH, argv[1]);
StringCchCat(szDir, MAX_PATH, L"\\*");

// Find the first file in the directory
hFind = FindFirstFile(szDir, &ffd);

if (INVALID_HANDLE_VALUE == hFind)
{
    DisplayErrorBox(L"FindFirstFile()");
    return dwError;
}

// List all the files in the directory with some info about them
do
{
    if (ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
    {
        wprintf(L" %s <DIR>\n", ffd.cFileName);
    }
    else
    {
        filesize.LowPart = ffd.nFileSizeLow;
        filesize.HighPart = ffd.nFileSizeHigh;
        wprintf(L" %s %ld bytes\n", ffd.cFileName,
filesize.QuadPart);
    }
}
while (FindNextFile(hFind, &ffd) != 0);

dwError = GetLastError();
if (dwError != ERROR_NO_MORE_FILES)
{
    DisplayErrorBox(L"FindFirstFile()");
}

FindClose(hFind);
return dwError;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
```

```

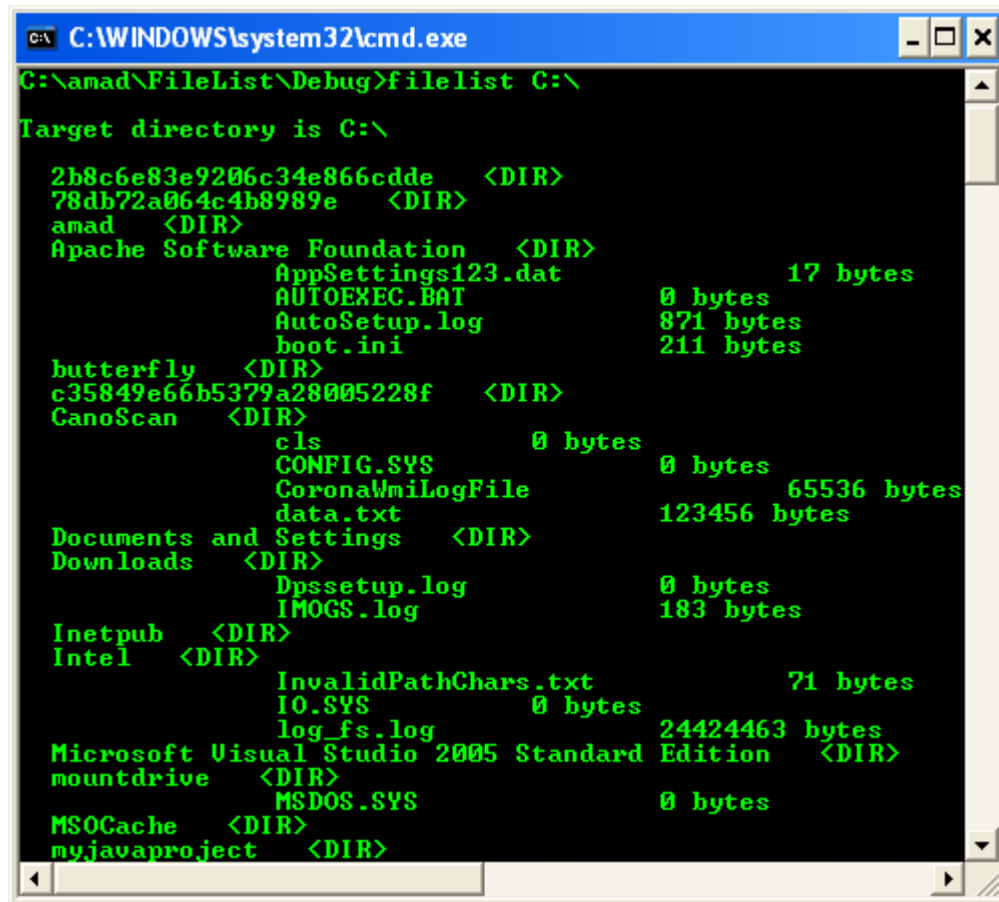
    FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
    (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR), L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}

```

Build and run the project. The following screenshots are sample outputs.



```

C:\WINDOWS\system32\cmd.exe
C:\amad\FileList\Debug>filelist C:\

Target directory is C:\

2b8c6e83e9206c34e866cdde <DIR>
78db72a064c4b8989e <DIR>
amad <DIR>
Apache Software Foundation <DIR>
    AppSettings123.dat 17 bytes
    AUTOEXEC.BAT 0 bytes
    AutoSetup.log 871 bytes
    boot.ini 211 bytes
butterfly <DIR>
c35849e66b5379a28005228f <DIR>
CanoScan <DIR>
    cls 0 bytes
    CONFIG.SYS 0 bytes
    CoronaWmiLogFile 65536 bytes
    data.txt 123456 bytes
Documents and Settings <DIR>
Downloads <DIR>
    Dpssetup.log 0 bytes
    IMOGS.log 183 bytes
Inetpub <DIR>
Intel <DIR>
    InvalidPathChars.txt 71 bytes
    IO.SYS 0 bytes
    log_fs.log 24424463 bytes
Microsoft Visual Studio 2005 Standard Edition <DIR>
mountdrive <DIR>
    MSDOS.SYS 0 bytes
MSOCache <DIR>
my.javaproject <DIR>

```

```

C:\WINDOWS\system32\cmd.exe

C:\amad\FileList\Debug>filelist C:\WINDOWS\system32\config
Target directory is C:\WINDOWS\system32\config

. <DIR>
.. <DIR>
AppEvent.Evt      524288 bytes
DEFAULT           524288 bytes
default.LOG       1024 bytes
default.sav       94208 bytes
Internet.evt      65536 bytes
MyCustLo.evt      65536 bytes
ODiag.evt         65536 bytes
OSession.evt      196608 bytes
SAM               262144 bytes
SAM.LOG           1024 bytes
SecEvent.Evt      65536 bytes
SECURITY          262144 bytes
SECURITY.LOG      1024 bytes
SOFTWARE          56885248 bytes
software.LOG      1024 bytes
software.sav      659456 bytes
SysEvent.Evt      524288 bytes
SYSTEM            7602176 bytes
system.LOG        8192 bytes
system.sav        876544 bytes
systemprofile     <DIR>
TempKey.LOG       1024 bytes
userdiff          262144 bytes
userdiff.LOG      1024 bytes

C:\amad\FileList\Debug>

```

```

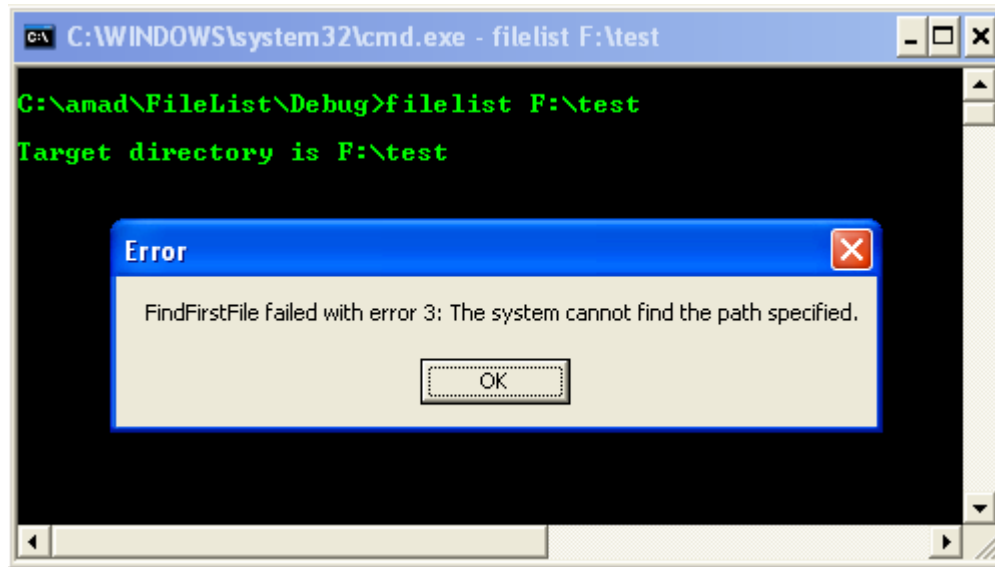
C:\WINDOWS\system32\cmd.exe

C:\amad\FileList\Debug>FileList "C:\Documents and Settings"
Target directory is C:\Documents and Settings

. <DIR>
.. <DIR>
Administrator <DIR>
All Users <DIR>
Default User <DIR>
LocalService <DIR>
mike spoon <DIR>
NetworkService <DIR>

C:\amad\FileList\Debug>

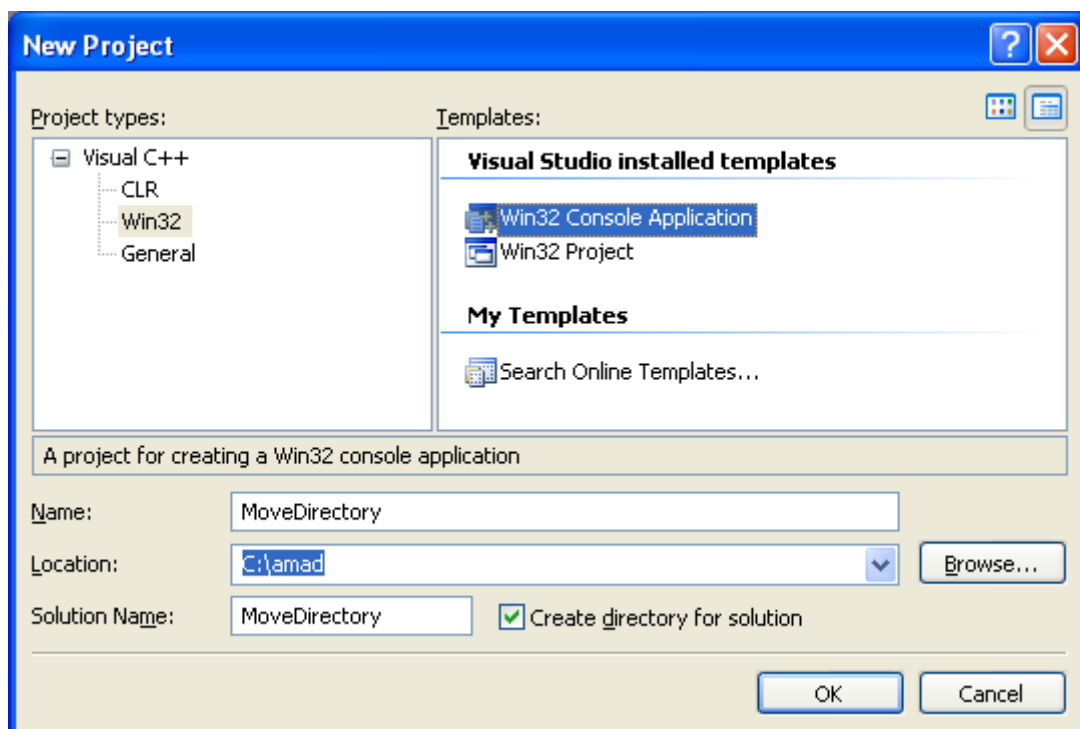
```



Moving Directories Program Example

To move a directory to another location, along with the files and subdirectories contained within it, call the `MoveFileEx()`, `MoveFileWithProgress()`, or `MoveFileTransacted()` function. The `MoveFileWithProgress()` function has the same functionality as `MoveFileEx()`, except that `MoveFileWithProgress()` enables you to specify a callback routine that receives notifications on the progress of the operation. The `MoveFileTransacted()` function enables you to perform the operation as a transacted operation. The following example demonstrates the use of the `MoveFileEx()` function with a directory.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.
Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
// Safer versions
#include <strsafe.h>

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int wmain(int argc, WCHAR *argv[])
{
    printf("\n");
    // Verify
    if( argc != 3 )
    {
        wprintf(L"Error: Incorrect number of argument!\n");
        wprintf(L"Description: Moves a directory and its contents\n");
        wprintf(L"Usage: %s [source_dir] [target_dir]\n", argv[0]);
        wprintf(L"Example: %s C:\\testdir C:\\newtestdir\n", argv[0]);
        wprintf(L"          The target directory must not exist.\n\n");
        return 1;
    }

    // Move the source directory to the target directory location.
    // The target directory must be on the same drive as the source.
    // The target directory cannot already exist
    if (!MoveFileEx(argv[1], argv[2], MOVEFILE_WRITE_THROUGH))
    {
        DisplayErrorBox(L"MoveFileEx()");
        return 1;
    }
    else wprintf(L"%s has been moved to %s\n", argv[1], argv[2]);
    return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

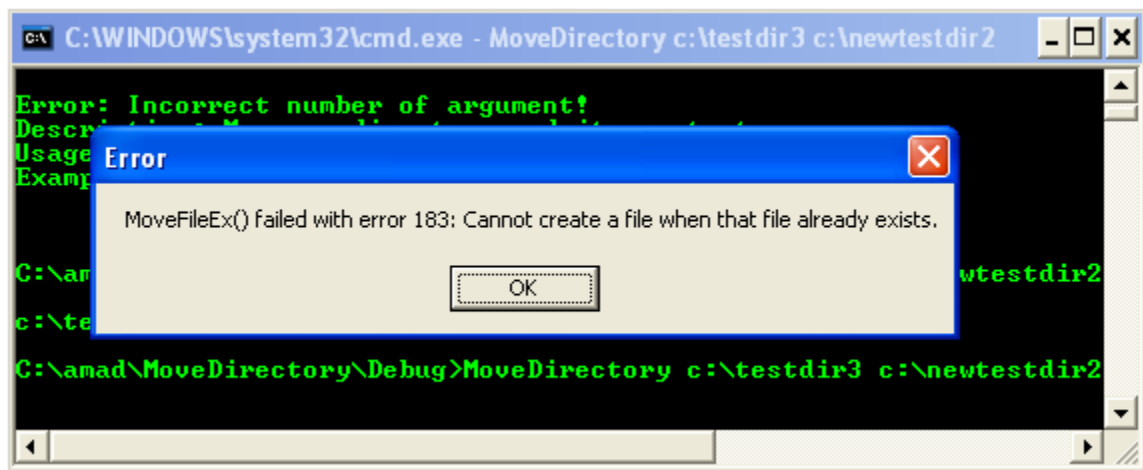
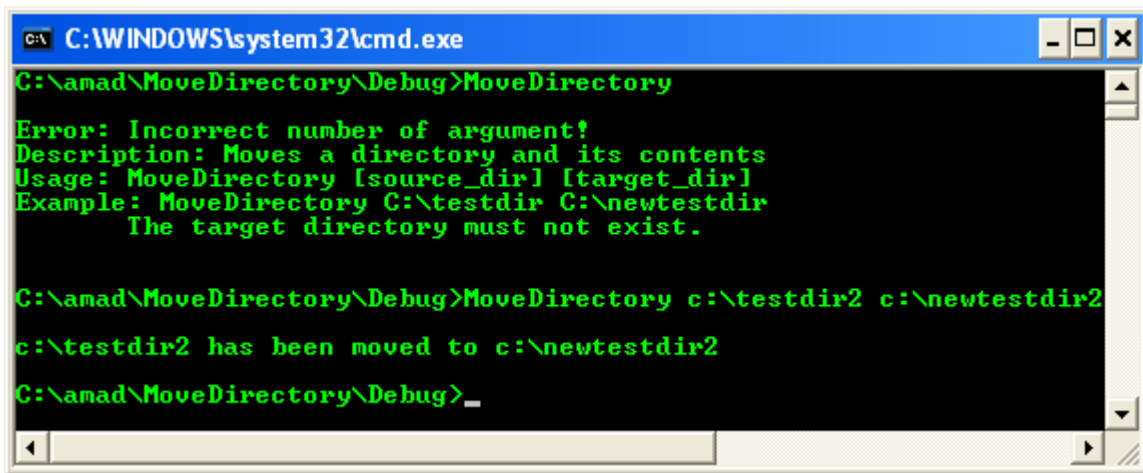
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
```

```
(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
StringCchPrintf((LPTSTR)lpDisplayBuf,
    LocalSize(lpDisplayBuf) / sizeof(WCHAR), L"%s failed with error %d:
%s",
    lpszFunction, dw, lpMsgBuf);
MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

LocalFree(lpMsgBuf);
LocalFree(lpDisplayBuf);
}
```

Build and run the project. The following screenshots are sample outputs.



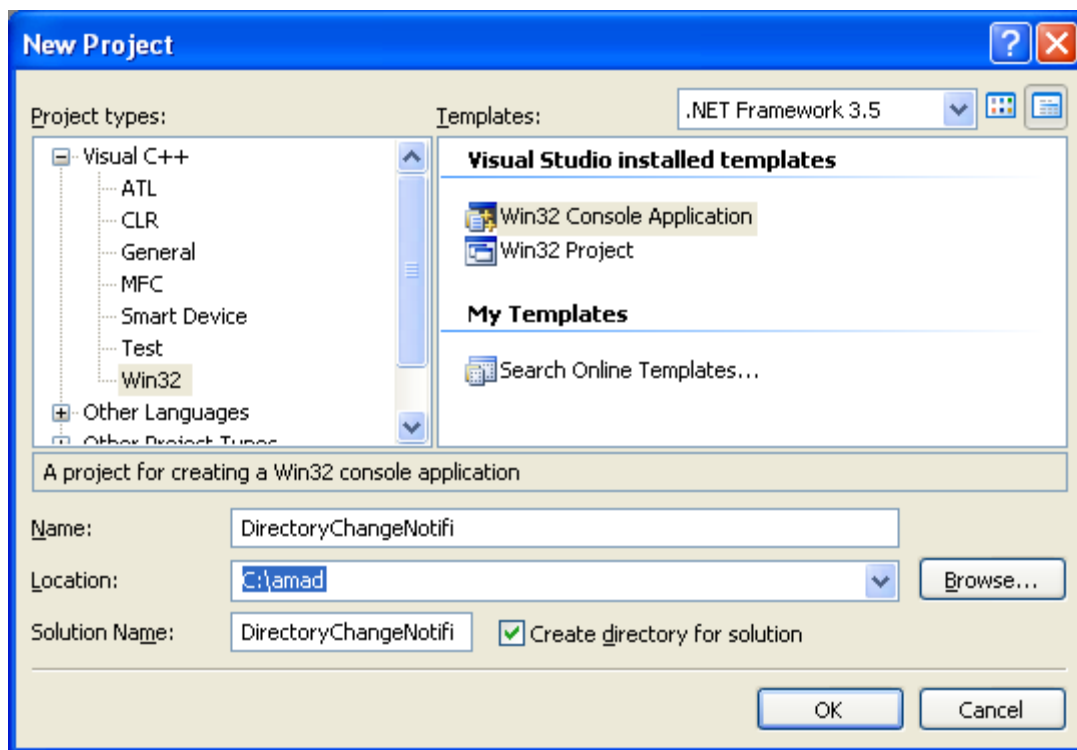
Obtaining the Directory Change Notifications Program Example

An application can monitor the contents of a directory and its subdirectories by using change notifications. Waiting for a change notification is similar to having a read operation pending against a directory and, if necessary, its subdirectories. When something changes within the directory being watched, the read operation is completed. For example, an application can use these functions to update a directory listing whenever a file name within the monitored directory changes.

An application can specify a set of conditions that trigger a change notification by using the `FindFirstChangeNotification()` function. The conditions include changes to file names, directory names, attributes, file size, time of last write, and security. This function also returns a handle that can be waited on by using the wait functions. If the wait condition is satisfied, `FindNextChangeNotification()` can be used to provide a notification handle to wait on subsequent changes. However, these functions do not indicate the actual change that satisfied the wait condition. Use `FindCloseChangeNotification()` to close the notification handle.

To retrieve information about the specific change as part of the notification, use the `ReadDirectoryChangesW()` function. This function also enables you to provide a completion routine. The following example monitors the directory tree for directory name changes. It also monitors a directory for file name changes. The example uses the `FindFirstChangeNotification()` function to create two notification handles and the `WaitForMultipleObjects()` function to wait on the handles. Whenever a directory is created or deleted in the tree, the example should update the entire directory tree. Whenever a file is created or deleted in the directory, the example should refresh the directory. This simplistic example uses the `ExitProcess()` function for termination and cleanup, but more complex applications should always use proper resource management such as `FindCloseChangeNotification()` where appropriate.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
// Safer versions
#include <strsafe.h>
```



```
// Function prototypes
// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);
void RefreshDirectory(LPTSTR);
void RefreshTree(LPTSTR);
void WatchDirectory(LPTSTR);

int wmain(int argc, WCHAR *argv[])
{
    if(argc != 2)
    {
        wprintf(L"Change notification for directory\n");
        wprintf(L"Usage: %s <dir>\n", argv[0]);
        return 1;
    }

    WatchDirectory(argv[1]);
    return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and clean up
    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
        (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(WCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(WCHAR), L"%s failed with error %d:
%s",
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}

void WatchDirectory(LPTSTR lpDir)
{
    DWORD dwWaitStatus;
    HANDLE dwChangeHandles[2];
    WCHAR lpDrive[4];
    WCHAR lpFile[_MAX_FNAME];
```

```
WCHAR lpExt[_MAX_EXT];

// Breaks a path name into components. These are versions of _splitpath(),
// _wsplitpath() with security enhancements
_wsplitpath_s(lpDir, lpDrive, 4, NULL, 0, lpFile, _MAX_FNAME, lpExt,
_MAX_EXT);

lpDrive[2] = '\\';
lpDrive[3] = '\\0';

// Watch the directory for file creation and deletion
dwChangeHandles[0] = FindFirstChangeNotification(
    lpDir,                      // directory to watch
    FALSE,                     // do not watch subtree
    FILE_NOTIFY_CHANGE_FILE_NAME); // watch file name changes

if (dwChangeHandles[0] == INVALID_HANDLE_VALUE)
{
    DisplayErrorBox(L"FindFirstChangeNotification() 1");
    ExitProcess(GetLastError());
}

// Watch the subtree for directory creation and deletion
dwChangeHandles[1] = FindFirstChangeNotification(
    lpDrive,                   // directory to watch
    TRUE,                     // watch the subtree
    FILE_NOTIFY_CHANGE_DIR_NAME); // watch dir name changes

if (dwChangeHandles[1] == INVALID_HANDLE_VALUE)
{
    DisplayErrorBox(L"FindFirstChangeNotification() 2");
    ExitProcess(GetLastError());
}

// Make a final validation check on our handles
if ((dwChangeHandles[0] == NULL) || (dwChangeHandles[1] == NULL))
{
    wprintf(L"\nFindFirstChangeNotification() - unexpected NULL.\n");
    ExitProcess(GetLastError());
}

// Change notification is set. Now wait on both notification
// handles and refresh accordingly
while (TRUE)
{
    // Wait for notification
    wprintf(L"\nWaiting for notification...\n");

    dwWaitStatus = WaitForMultipleObjects(2, dwChangeHandles, FALSE,
INFINITE);

    switch (dwWaitStatus)
    {
        case WAIT_OBJECT_0:
            // A file was created, renamed, or deleted in the directory.
            // Refresh this directory and restart the notification
            RefreshDirectory(lpDir);
            if (FindNextChangeNotification(dwChangeHandles[0]) == FALSE )
            {

```

```

        DisplayErrorBox(L"FindNextChangeNotification()");
        ExitProcess(GetLastError());
    }
    break;

case WAIT_OBJECT_0 + 1:
    // A directory was created, renamed, or deleted.
    // Refresh the tree and restart the notification
    RefreshTree(lpDrive);
    if (FindNextChangeNotification(dwChangeHandles[1]) == FALSE )
    {
        DisplayErrorBox(L"FindNextChangeNotification()");
        ExitProcess(GetLastError());
    }
    break;

case WAIT_TIMEOUT:
    // A timeout occurred, this would happen if some value other
    // than INFINITE is used in the Wait call and no changes
occur.
    // In a single-threaded environment you might not want an
INFINITE wait
    wprintf(L"\nNo changes in the timeout period.\n");
    break;

default:
    wprintf(L"\n ERROR: Unhandled dwWaitStatus.\n");
    ExitProcess(GetLastError());
    break;
    }
}

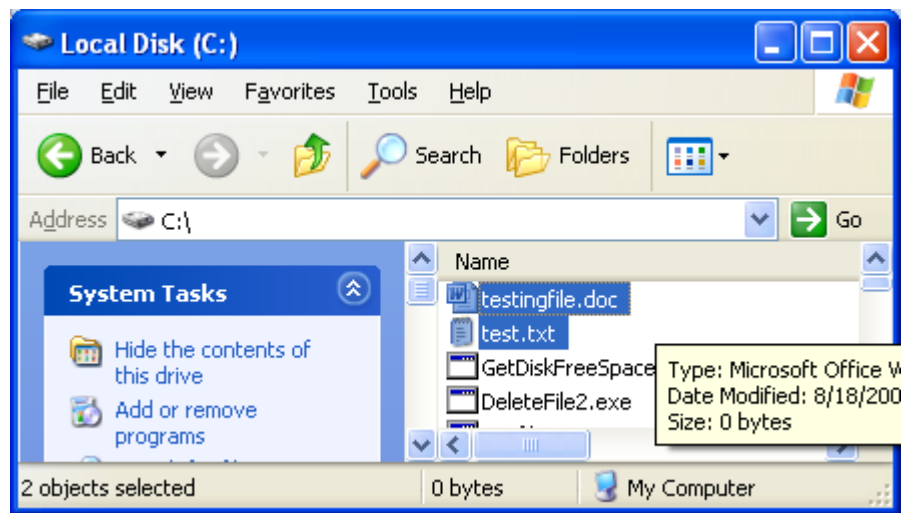
void RefreshDirectory(LPTSTR lpDir)
{
    // This is where you might place code to refresh your
    // directory listing, but not the subtree because it
    // would not be necessary
    wprintf(L"Directory %s changed!\n", lpDir);
}

void RefreshTree(LPTSTR lpDrive)
{
    // This is where you might place code to refresh your
    // directory listing, including the subtree.
    wprintf(L"Directory tree %s changed!\n", lpDrive);
}

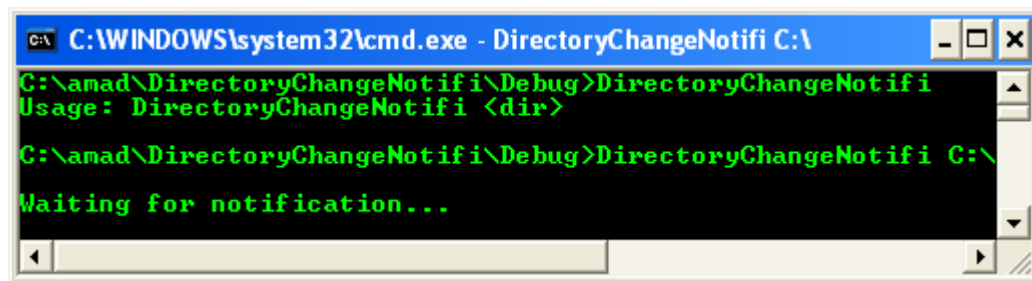
```

Build the project.

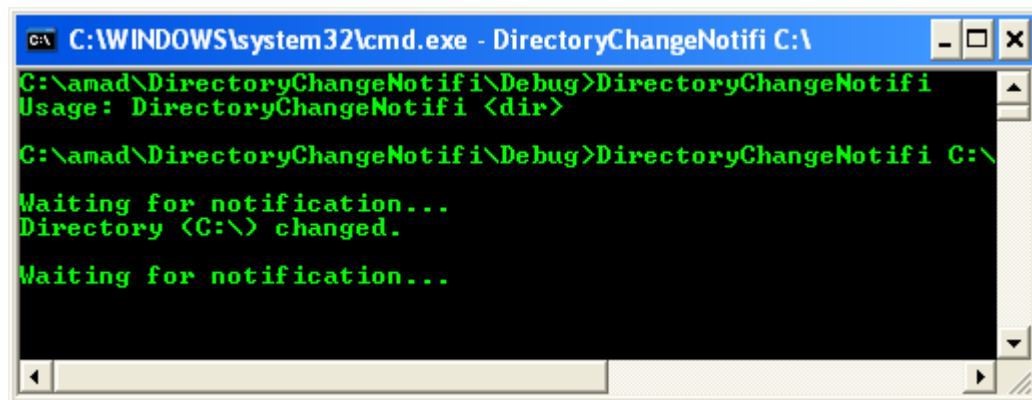
Create several empty text or other files in any folders of C drive. For example, two files was created as shown below in C:\



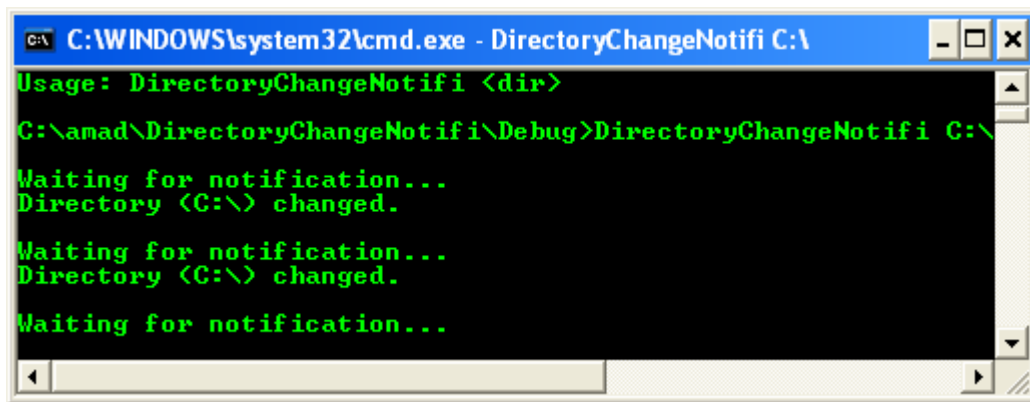
Run the program.



By leaving the output console, rename test.txt to test1.txt. Notice the output.

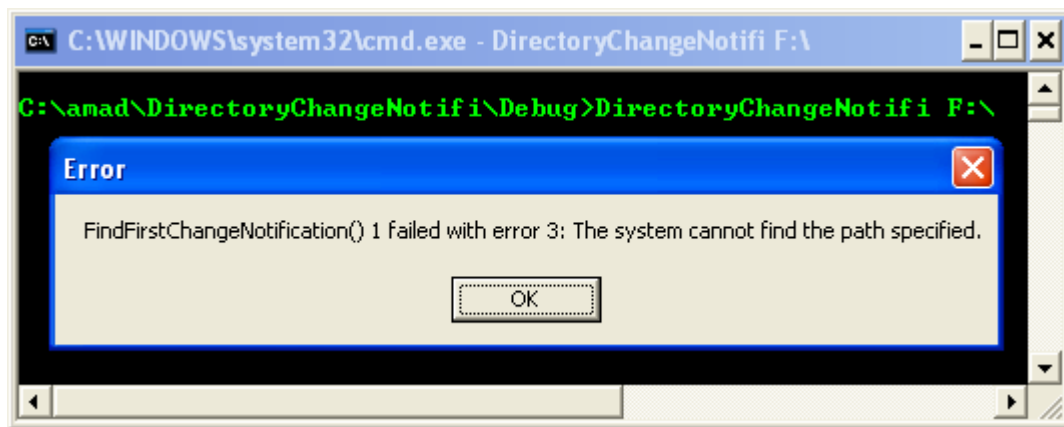


Then delete the testingfile.doc file. Notice the output.



```
C:\WINDOWS\system32\cmd.exe - DirectoryChangeNotifi C:\
Usage: DirectoryChangeNotifi <dir>
C:\namad\DirectoryChangeNotifi\Debug>DirectoryChangeNotifi C:\
Waiting for notification...
Directory (C:\) changed.
Waiting for notification...
Directory (C:\) changed.
Waiting for notification...
```

When there is an error, the output gives some useful meaning.



Retrieving and Changing File Attributes Program Example

An application can retrieve the file attributes by using the `GetFileAttributes()` or `GetFileAttributesEx()` function. The `CreateFile()` and `SetFileAttributes()` functions can set many of the attributes. However, applications cannot set all attributes.

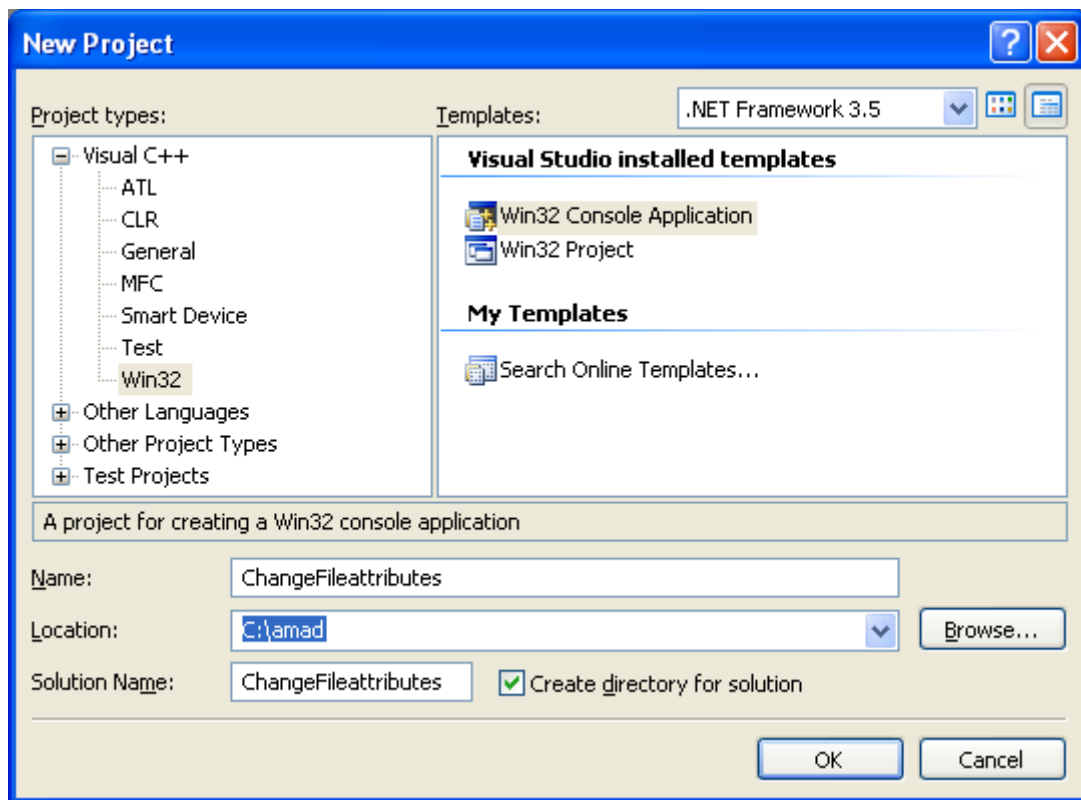
The following code example uses the `CopyFile()` function to copy all text files (.txt) in the current directory to a new directory of read-only files. Files in the new directory are changed to read only, if necessary.

The application creates the directory specified as a parameter by using the `CreateDirectory()` function. The directory must not exist already.

The application searches the current directory for all text files by using the `FindFirstFile()` and `FindNextFile()` functions. Each text file is copied to the `\TextRO` directory. After a file is copied, the `GetFileAttributes()` function determines whether or not a file is read only. If the file is not read only, the application changes directories to `\TextRO` and converts the copied file to read only by using the `SetFileAttributes()` function.

After all text files in the current directory are copied, the application closes the search handle by using the `FindClose()` function.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.

Next, add the following source code.

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

// #define BUFSIZE MAX_PATH

// A prototype that receives a function name, displaying
// system error code and its respective message
void DisplayErrorBox(LPTSTR lpszFunction);

int wmain(int argc, WCHAR* argv[])
{
    WIN32_FIND_DATA FileData;
    HANDLE hSearch;
    DWORD dwAttrs; //, dwRet;
    WCHAR szNewPath[MAX_PATH];
    // WCHAR Buffer[BUFSIZE];

    BOOL fFinished = FALSE;

    if(argc != 2)
    {
        wprintf(L"Copying text files in the current directory to a new
directory\n");
        wprintf(L"Usage: %s <new dir>\n", argv[0]);
    }
}
```

```
wprintf(L"Example: %s C:\\testdir\\n", argv[0]);
return 1;
}

// Create a new directory
if (!CreateDirectory(argv[1], NULL))
{
    DisplayErrorBox(L"CreateDirectory()");
    return 1;
}
else
    wprintf(L"%s directory was successfully created!\\n", argv[1]);

// Start searching for text files in the current directory
hSearch = FindFirstFile(L"*.txt", &FileData);
if (hSearch == INVALID_HANDLE_VALUE)
{
    wprintf(L"No text files found.\\n");
    return 1;
}

// Copy each .TXT file to the new directory
// and change it to read only, if not already
while (!fFinished)
{
    StringCchPrintf(szNewPath, MAX_PATH, L"%s\\%s", argv[1],
FileData.cFileName);

    if (CopyFile(FileData.cFileName, szNewPath, FALSE))
    {
        dwAttrs = GetFileAttributes(FileData.cFileName);
        if (dwAttrs==INVALID_FILE_ATTRIBUTES)
            return 1;

        if (!(dwAttrs & FILE_ATTRIBUTE_READONLY))
        {
            SetFileAttributes(szNewPath, dwAttrs | FILE_ATTRIBUTE_READONLY);
        }
    }
    else
    {
        wprintf(L"Could not copy file.\\n");
        return 1;
    }

    if (!FindNextFile(hSearch, &FileData))
    {
        if (GetLastError() == ERROR_NO_MORE_FILES)
        {
            wprintf(L"Copied *.txt to %s directory\\n", argv[1]);
            fFinished = TRUE;
        }
        else
        {
            wprintf(L"Could not find next file.\\n");
            return 1;
        }
    }
}
}
```

```
// Close the search handle
FindClose(hSearch);

/* Optional extra codes
// Change to the new created directory
if(!SetCurrentDirectory(argv[1]))
{
    DisplayErrorBox(L"SetCurrentDirectory()");
    return 1;
}
wprintf(L"Set current directory to %s\n", argv[1]);

    dwRet = GetCurrentDirectory(BUFSIZE, Buffer);

if( dwRet == 0 )
{
    DisplayErrorBox(L"GetCurrentDirectory()");
    return 1;
}

if(dwRet > BUFSIZE)
{
    printf("Buffer too small; need %d characters\n", dwRet);
    return 1;
}

wprintf(L"Current directory is %s\n", Buffer);

// ===== Need another loop, find first, find next =====
// Removing the read only attribute of those files
// TODO: Change all the read only file attribute to write
//         so that all those file can be deleted

// Deleting all the files
if(DeleteFile(L"*.txt") == 0)
    DisplayErrorBox(L"DeleteFile()");
// ===== End loop, find first, find next =====

// Then, when the directory is empty we can
// remove it
if(RemoveDirectory(argv[1]) == 0)
{
    DisplayErrorBox(L"RemoveDirectory()");
    wprintf(L"%s failed to be removed!\n", argv[1]);
}

    */
return 0;
}

void DisplayErrorBox(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
```



```

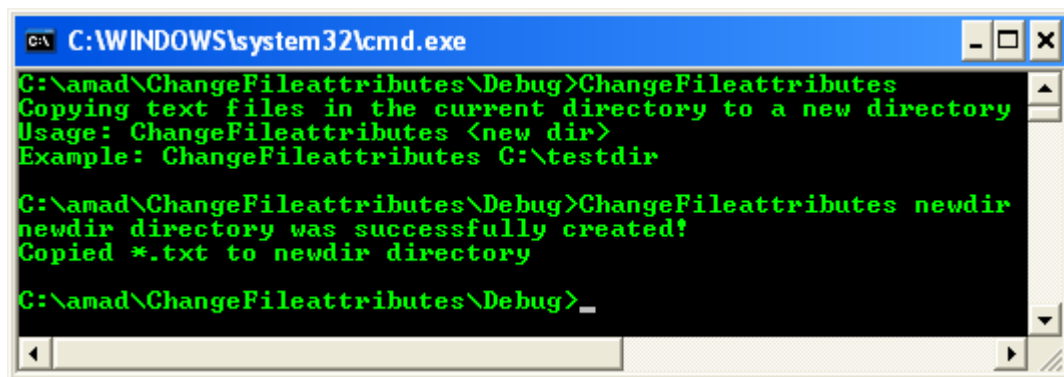
FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
NULL,
dw,
MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
(LPTSTR) &lpMsgBuf,
0, NULL );

// Display the error message and clean up
lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpzFunction)+40)*sizeof(WCHAR));
StringCchPrintf((LPTSTR)lpDisplayBuf,
    LocalSize(lpDisplayBuf) / sizeof(WCHAR), L"%s failed with error %d:
%s",
    lpzFunction, dw, lpMsgBuf);
MessageBox(NULL, (LPCTSTR)lpDisplayBuf, L"Error", MB_OK);

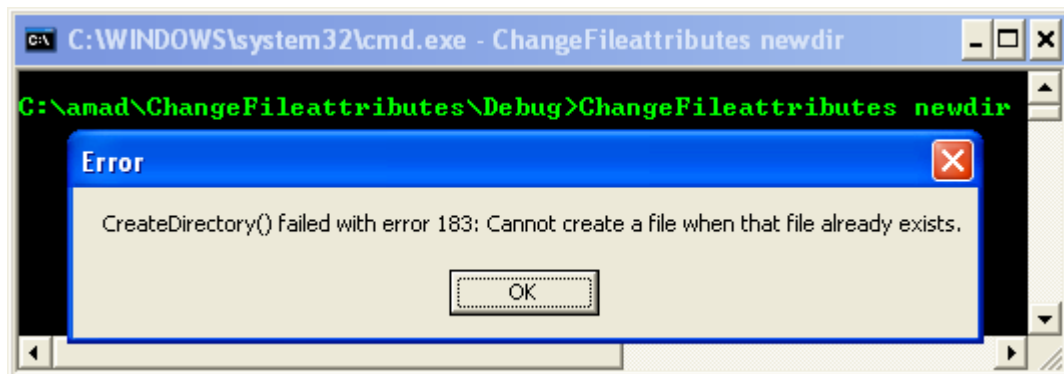
LocalFree(lpMsgBuf);
LocalFree(lpDisplayBuf);
}

```

Build the project and to test this program you need to create a few text files under the project's Debug folder. Then, run the project. The following screenshots are sample outputs.



The following screenshot shows a sample output when the new directory already existed.



The following sample output generated when there is no text file found.

```

C:\WINDOWS\system32\cmd.exe

C:\amad\ChangeFileattributes\Debug>ChangeFileattributes newdir

C:\amad\ChangeFileattributes\Debug>ChangeFileattributes C:\anotherdir
C:\anotherdir directory was successfully created!
No text files found.

C:\amad\ChangeFileattributes\Debug>
  
```

Directory Management Reference

The following elements are used in directory management:

1. Directory Management Control Codes
2. Directory Management Functions
3. Directory Management Macros
4. Directory Management Structures

Directory Management Control Codes

The following control codes are used with file compression and decompression.

Value	Meaning
FSCTL_GET_COMPRESSION	Obtains the compression state of a file or directory
FSCTL_SET_COMPRESSION	Sets the compression state of a file or directory.

The following control codes are used with reparse points.

Value	Meaning
FSCTL_DELETE_REPARSE_POINT	Deletes a reparse point for a file or directory.
FSCTL_GET_REPARSE_POINT	Returns reparse point data for a file or directory.
FSCTL_SET_REPARSE_POINT	Sets a reparse point on a file or directory.

Directory Management Functions

The following functions are used in directory management.

Function	Description
CreateDirectory()	Creates a new directory.
CreateDirectoryEx()	Creates a new directory with the attributes of a specified template directory.
CreateDirectoryTransacted()	Creates a new directory as a transacted operation, with the attributes of a specified template directory.
CreateFile()	Opens an existing directory object.
DeleteFile()	Deletes an existing directory.
FindCloseChangeNotification()	Stops change notification handle monitoring.
FindFirstChangeNotification()	Creates a change notification handle.

FindFirstFile()	Searches a directory for a file or subdirectory whose name matches the specified name.
FindFirstFileEx()	Searches a directory for a file or subdirectory whose name and attributes match those specified.
FindNextChangeNotification()	Requests that the operating system signal a change notification handle the next time it detects an appropriate change.
FindNextFile()	Continues a file search.
GetCurrentDirectory()	Retrieves the current directory for the current process.
MoveFile()	Moves an existing directory.
MoveFileEx()	Moves an existing directory.
ReadDirectoryChangesW()	Retrieves information describing the changes occurring within a directory.
RemoveDirectory()	Deletes an existing empty directory.
RemoveDirectoryTransacted()	Deletes an existing empty directory as a transacted operation.
SetCurrentDirectory()	Changes the current directory for the current process.

Directory Management Macros

The following macros are used with reparse points:

1. IsReparseTagMicrosoft() - Determines whether a reparse point tag indicates a Microsoft reparse point.
2. IsReparseTagNameSurrogate() - Determines whether a tag's associated reparse point is a surrogate for another named entity (for example, a mounted folder).

Directory Management Structures

The following structures are used in directory management:

1. FILE_NOTIFY_INFORMATION - Describes the changes found by the ReadDirectoryChangesW() function.
2. REPARSE_GUID_DATA_BUFFER - Contains information about a reparse point. It is used by the FSCTL_GET_REPARSE_POINT control code.