Win32 Windows Volume Program and Code Example

What do we have in this session?

Deleting a Mounted Folder Program Example
Windows Master File Table (MFT)
Master File Table Program Example 1
Master File Table Program Example 2: Reading and Dumping the Deleted Files
Master File Table Program Example 3: Using Non-Windows Types
Listing the Deleted Files from Master File Table (MFT)
Another Day, Another MFT Program Example: List, Recover and Delete the Deleted Files from Master File Table
Windows Master Boot Record (MBR)
Volume Management Reference
- **Volume Management Functions**
- **Volume Management Control Codes**
- **Volume Management Structures**

Optional Pre-requirement

For the beginners, you may want to learn about the motherboard chipsets and memory map and kernel boot sequence of the operating system. The information can be found at the following URLs:

1. Motherboard chipset and memory map.
2. How computer boot-up.
3. The kernel boot process.

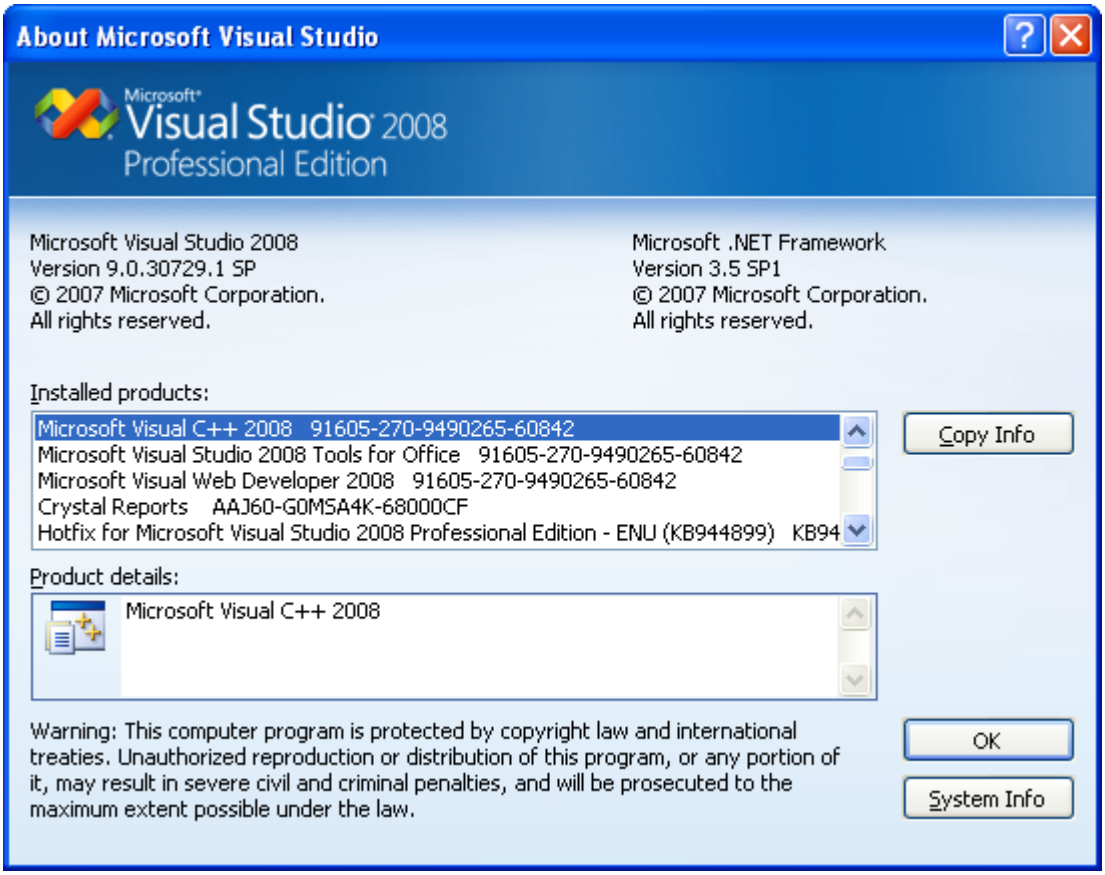
Some Notes to Students

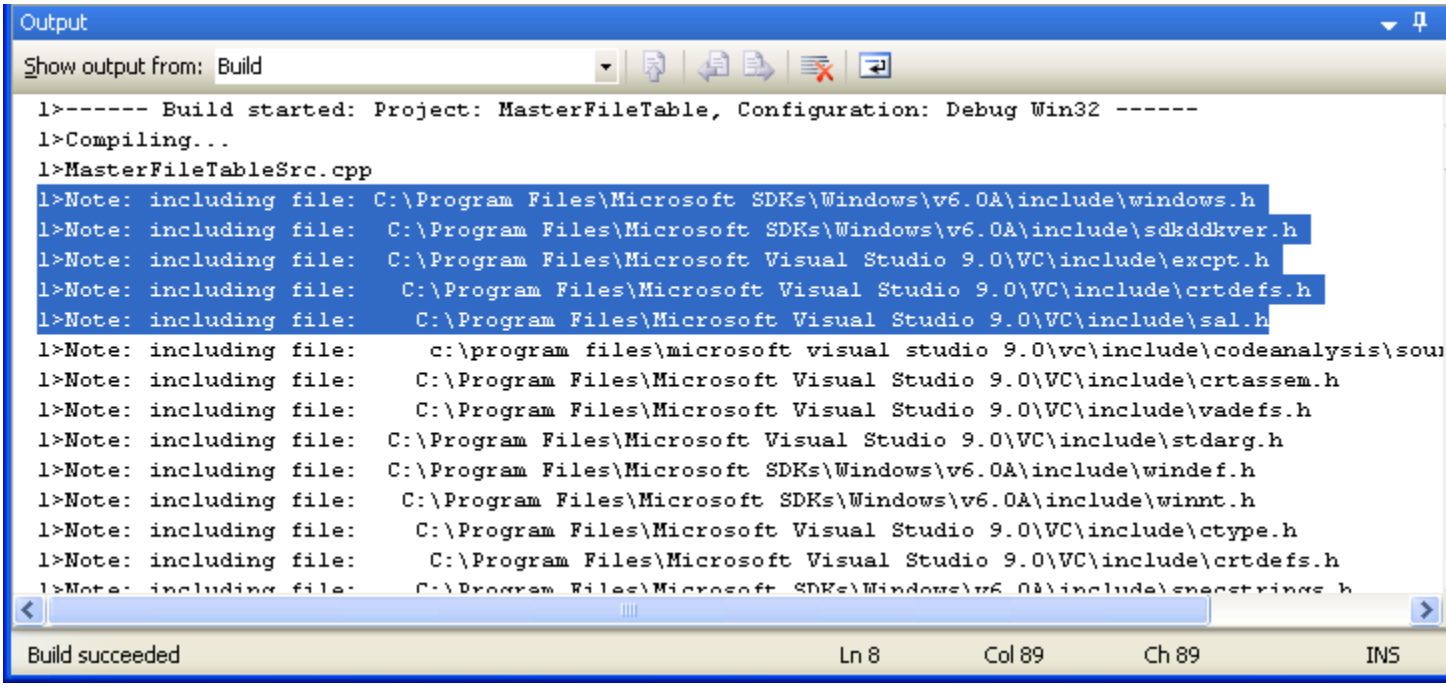
The knowledge and skill that you are supposed to acquire are several C/C++ programming language constructs such as:

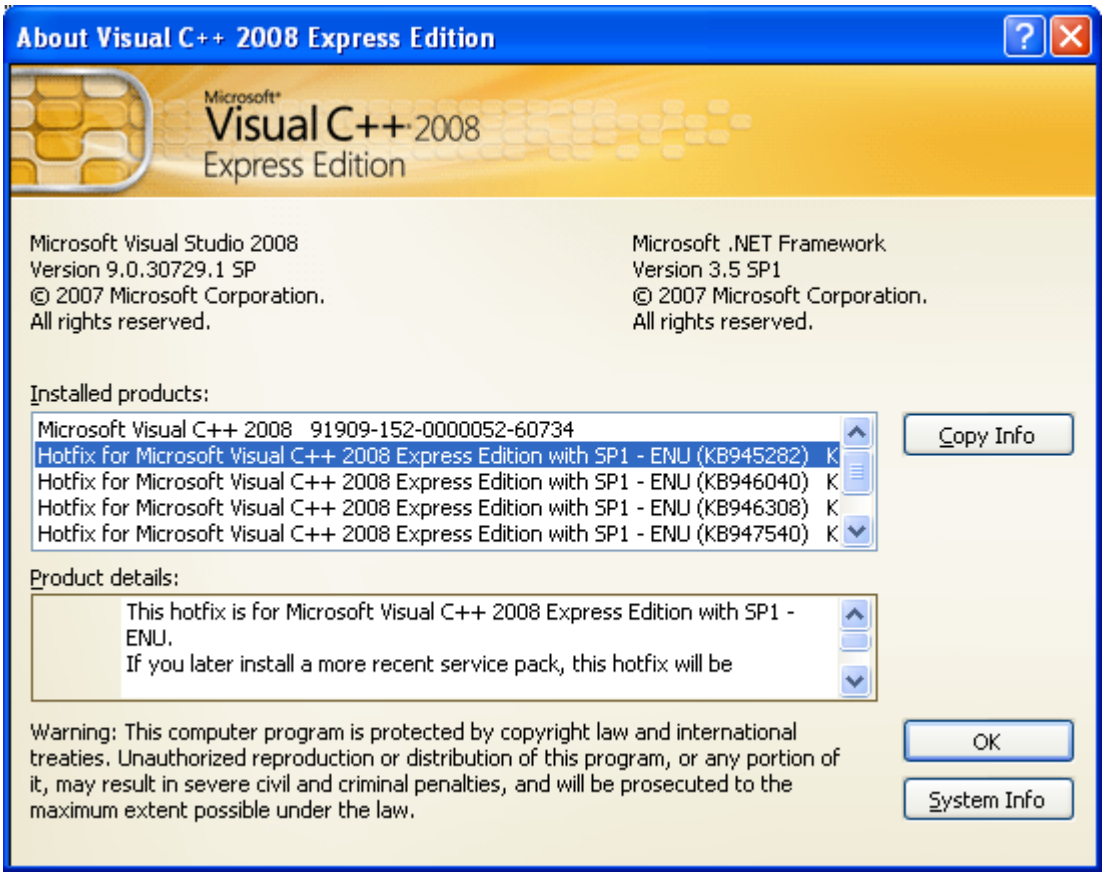
1. (C/C++) Windows Data Types.
2. Typedef.
3. Enumerations.
4. Functions.
5. Structures.
6. Constants (such as used in Control Codes).
7. Macros.
8. The related header files to be included in the program.
9. The related libraries to be linked.
10. Familiarization with Visual Studio 2008 Express Edition/Visual Studio 2008

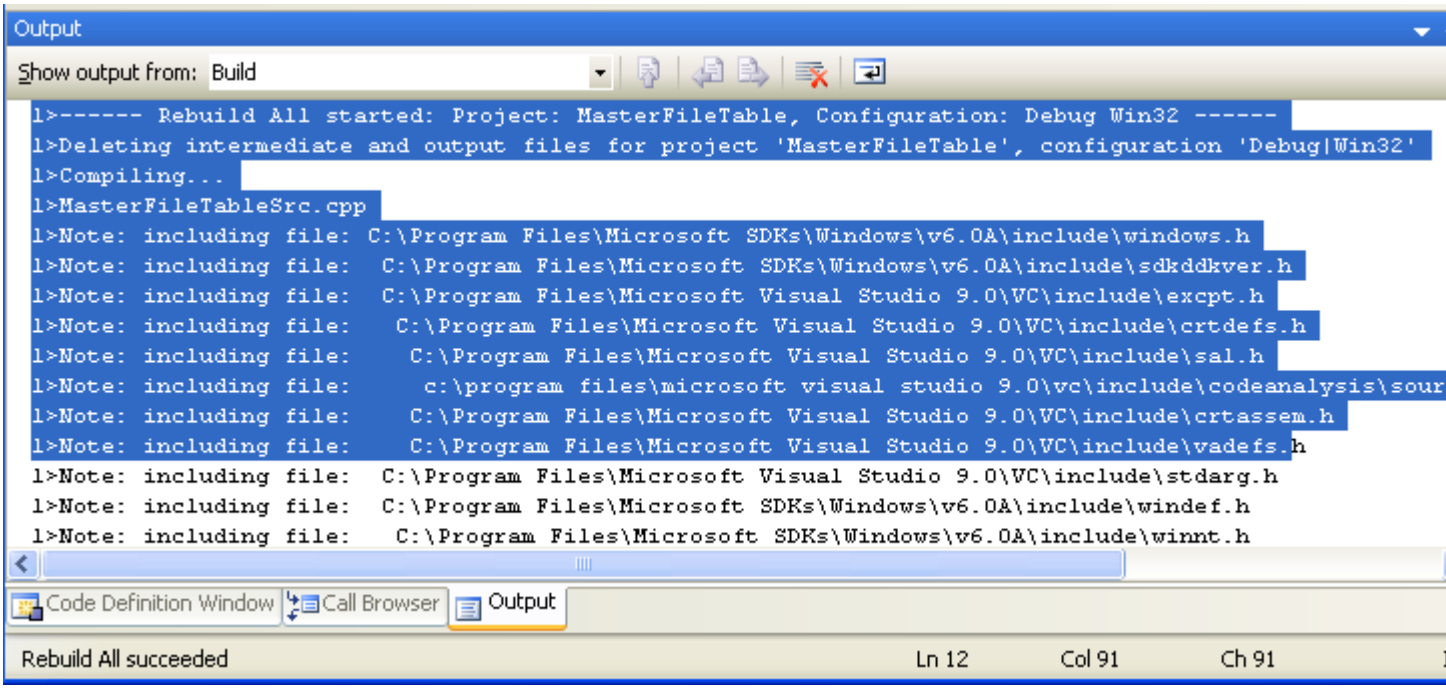
Environment for the Program Example (Build and Run)

All the program examples build using Visual Studio 2008/210 with SP/Visual Studio 2008 Express Edition (free) with SP (C/C++) with PSDK installed and run on Windows XP Pro SP2. All the projects are empty Win32 console mode applications, using default settings and source codes are in C/C++. The Visual Studio and PSDK versions are shown in the following screenshots.




Output
Show output from: Build
1>------ Build started: Project: MasterFileTable, Configuration: Debug Win32 ------
1>Compiling...
1>MasterFileTableSrc.cpp
1>Note: including file: C:\Program Files\Microsoft SDKs\Windows\v6.0A\include\windows.h
1>Note: including file: C:\Program Files\Microsoft SDKs\Windows\v6.0A\include\sdkddkver.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\excpt.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\crtdefs.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\sal.h
1>Note: including file: c:\program files\microsoft visual studio 9.0\vc\include\codeanalysis\sou
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\crtassem.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\vadefs.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\stdarg.h
1>Note: including file: C:\Program Files\Microsoft SDKs\Windows\v6.0A\include\windef.h
1>Note: including file: C:\Program Files\Microsoft SDKs\Windows\v6.0A\include\winnt.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\ctype.h
1>Note: including file: C:\Program Files\Microsoft Visual Studio 9.0\VC\include\crtdefs.h
Build succeeded
Ln 8
Col 89
Ch 89
INS


About Visual C++ 2008 Express Edition
Microsoft
Visual C++ 2008
Express Edition
Microsoft Visual Studio 2008
Version 9.0.30729.1 SP
© 2007 Microsoft Corporation.
All rights reserved.
Microsoft .NET Framework
Version 3.5 SP1
© 2007 Microsoft Corporation.
All rights reserved.
Installed products:
Microsoft Visual C++ 2008 91909-152-0000052-60734
Hotfix for Microsoft Visual C++ 2008 Express Edition with SP1 - ENU (KB945282)
Hotfix for Microsoft Visual C++ 2008 Express Edition with SP1 - ENU (KB946040)
Hotfix for Microsoft Visual C++ 2008 Express Edition with SP1 - ENU (KB946308)
Hotfix for Microsoft Visual C++ 2008 Express Edition with SP1 - ENU (KB947540)
Copy Info
Product details:
This hotfix is for Microsoft Visual C++ 2008 Express Edition with SP1 - ENU.
If you later install a more recent service pack, this hotfix will be
Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.
OK
System Info



Brief Introduction

The highest level of organization in the file system is the volume. A file system resides on a volume. A volume contains **at least one partition**, which is a logical division of a physical disk. A volume that contains data that exists on one partition is called a **simple volume**, and a volume that contains data that exists on more than one partition is called a **multipartition volume**.
Volumes are implemented by a device driver called a volume manager. Examples include the FtDisk Manager, the Logical Disk Manager (LDM), and the VERITAS Logical Volume Manager (LVM). Volume managers provide a layer of physical abstraction, data protection (using some form of RAID), and performance.

File System Recognition

The goal of file system recognition is to allow the Windows operating system to have an additional option for a valid but unrecognized file system other than "RAW". To achieve this, **beginning with Windows 7 and Windows Server 2008 R2**, the system defines a fixed data structure type that can be written to the media on which an enabled technology that alters the file system format is active. This data structure, if present on logical disk sector zero, would then be recognized by the operating system and notify the user that the media contains a valid but unrecognized file system and is not a RAW volume if the drivers for the file system are not installed.

File System Recognition Components and Use

Several recent storage technologies have altered the on-disk file system format such that the media on which these technologies are enabled become unrecognizable to earlier versions of Windows due to the file system drivers not existing when a particular earlier version of Windows was released.

The previous default behavior in this scenario was as follows. When storage media is not a known file system, it is identified as RAW, and then propagated to the Windows Shell, where Autoplay prompts with the format user interface (UI). File system recognition can solve this if the authors of the new file system correctly write the proper data structure to the disk.
File system recognition uses the following components and layers within the operating system to achieve its goals:

1. Storage media, where a fixed data structure resides as a sequence of bytes arranged internally in a predefined structure called the **FILE_SYSTEM_RECOGNITION_STRUCTURE** data structure. It is the responsibility of the file system developer to create this on-disk structure properly.
2. File system recognition at the application level, achieved via the use of the **FSCTL_QUERY_FILE_SYSTEM_RECOGNITION** device I/O control code.
3. The Windows Shell UI uses the previously listed components to provide more flexible and robust Autoplay and related support for unrecognized file systems, but it can work only if the **FILE_SYSTEM_RECOGNITION_STRUCTURE** data structure exists in logical disk sector zero. Developers implementing new file systems should utilize this system to ensure that their file system is not mistakenly assumed to be of type "RAW".

Computing a File System Recognition Checksum Code Snippet

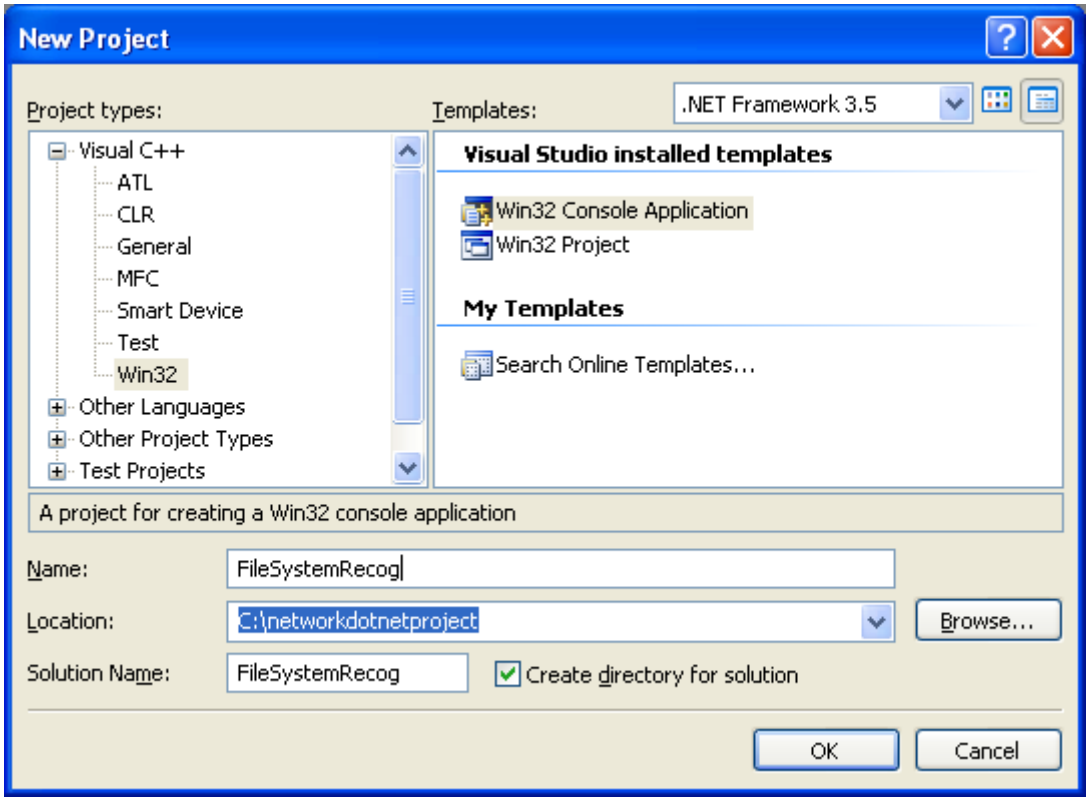
The FILE_SYSTEM_RECOGNITION_STRUCTURE structure, defined internally by Windows and used by file system recognition (FRS), contains a checksum value that must be properly computed for FRS to work properly with a specified unrecognized file system. The following code snippet accomplishes this computation (this code intended for vendors).

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

typedef struct _FILE_SYSTEM_RECOGNITION_STRUCTURE {
  UCHAR  Jmp[3];
  UCHAR  FsName[8];
  UCHAR  MustBeZero[5];
  ULONG  Identifier;
  USHORT Length;
  USHORT Checksum;
} FILE_SYSTEM_RECOGNITION_STRUCTURE, *PFILE_SYSTEM_RECOGNITION_STRUCTURE;

/*
Routine Description: This routine computes the file record checksum.
Arguments: Fsrs - Pointer to the record.
Return Value: The checksum result.
*/
USHORT ComputeFileSystemInformationChecksum(PFILE_SYSTEM_RECOGNITION_STRUCTURE
Fsrs)
{
    USHORT Checksum = 0;
    USHORT i;
    PUCHAR Buffer = (PUCHAR)Fsrs;
    USHORT StartOffset;
```

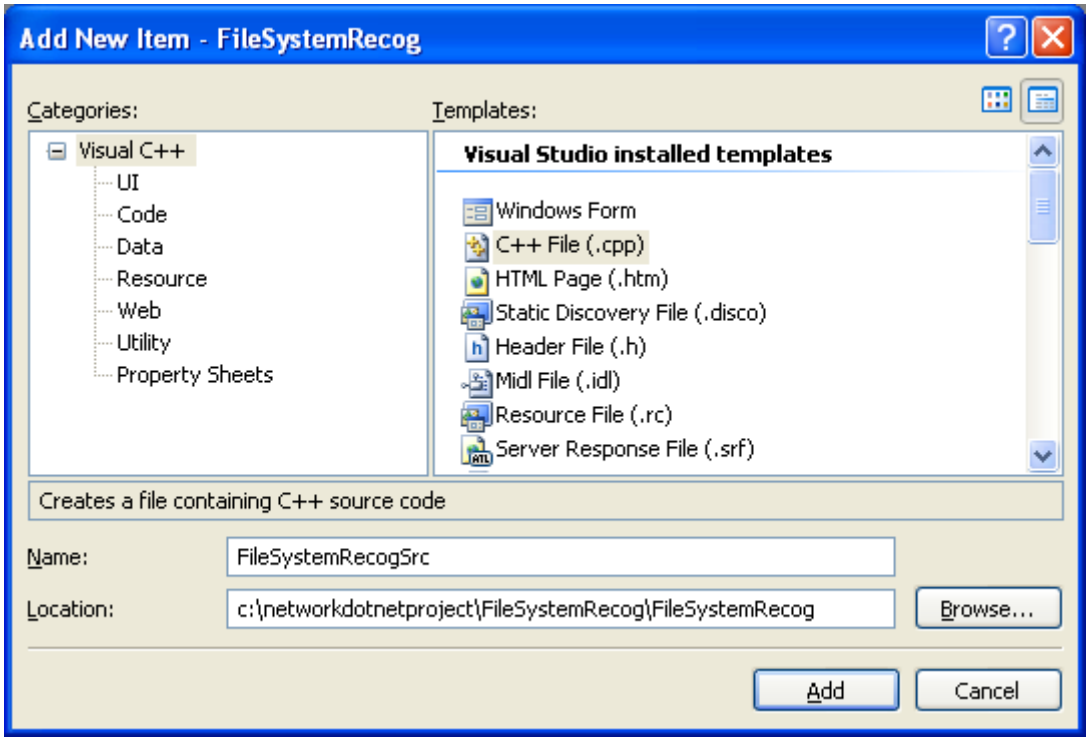
```
    //  Skip the jump instruction
    StartOffset = FIELD_OFFSET(FILE_SYSTEM_RECOGNITION_STRUCTURE, FsName);

      wprintf(L"In ComputeFileSystemInformationChecksum()\n");

    for (i = StartOffset; i < Fsrs->Length; i++)
      {
        //  Skip the checksum field itself, which is a USHORT.
        if ((i == FIELD_OFFSET(FILE_SYSTEM_RECOGNITION_STRUCTURE, Checksum)) ||
            (i == FIELD_OFFSET(FILE_SYSTEM_RECOGNITION_STRUCTURE, Checksum)+1))
            {
            continue;
        }
        Checksum = ((Checksum & 1) ? 0x8000 : 0) + (Checksum >> 1) + Buffer[i];
    }

    return Checksum;
}

int wmain()
{
      PFILE_SYSTEM_RECOGNITION_STRUCTURE Fsrs = {0};

      ComputeFileSystemInformationChecksum(Fsrs);
      return 0;
}
```

Obtaining File System Recognition Information Example

File system recognition is the ability to recognize storage media that contain a valid file system/volume layout that hasn't been defined yet, but the media is able to identify itself through the presence of the recognition structure defined internally by Windows.
Because no existing file system will recognize a new disk layout, the "RAW" file system will mount the volume and provide direct block level access. The "RAW" file system, incorporated in ntoskrnl, will have the ability to read the file system recognition structure and provide applications access to such structures through the file system control request FSCTL_QUERY_FILE_SYSTEM_RECOGNITION, shown in the following code snippet.
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

```
#include <windows.h>
#include <stdio.h>
#include <wchar.h>
// Min client - Windows 7;
// Min server - Windows Server 2008 R2
#include <Winioctl.h>

typedef struct _FILE_SYSTEM_RECOGNITION_INFORMATION {
  CHAR FileSystem[9];
}FILE_SYSTEM_RECOGNITION_INFORMATION, *PFILE_SYSTEM_RECOGNITION_INFORMATION;

STDMETHODIMP Check(PCWSTR  pcwszDrive)
{
    HRESULT                                 hr = S_OK;
    HANDLE                                  hDisk = INVALID_HANDLE_VALUE;
    FILE_SYSTEM_RECOGNITION_INFORMATION     FsRi = {0};
      DWORD dwIoControlCode = 0;
    BOOL                                    fResult = FALSE;
    ULONG                                   BytesReturned = 0;

    // Open the target, for example "\\.\F:"
    wprintf(L"CreateFile() on %s...\n", pcwszDrive);
    hDisk = CreateFile(pcwszDrive,
                       FILE_READ_ATTRIBUTES|SYNCHRONIZE|FILE_TRAVERSE,
                       FILE_SHARE_READ|FILE_SHARE_WRITE,
                       NULL, OPEN_EXISTING, 0, NULL);
    if(hDisk == INVALID_HANDLE_VALUE)
    {
        hr = GetLastError();
        wprintf(L"CreateFile() failed on %s, error = %u\n", pcwszDrive,
GetLastError());
        exit(1);
    }

    wprintf(L"CreateFile() succeeded.\n");
    wprintf(L"\nPress Any Key to send down the FSCTL...\n");
    _getwch();

    // Send down the FSCTL
    wprintf( L"Calling
DeviceIoControl(...,FSCTL_QUERY_FILE_SYSTEM_RECOGNITION,...)\n" );
      // http://msdn.microsoft.com/en-us/library/aa365729%28VS.85%29.aspx
      // and another one: http://msdn.microsoft.com/en-
us/library/cc232013%28PROT.10%29.aspx
    fResult = DeviceIoControl(hDisk, FSCTL_QUERY_FILE_SYSTEM_RECOGNITION, NULL,
0,
            &FsRi, sizeof(FsRi), &BytesReturned, NULL );
    if(!fResult)
    {
        hr = HRESULT_FROM_WIN32(GetLastError());
        wprintf(L"DeviceIoControl() failed, error %u\n", GetLastError());
        exit(1);
    }

    wprintf(L"DeviceIoControl() succeeded.\n");
```

```
    wprintf(L"FSCTL_QUERY_FILE_SYSTEM_RECOGNITION returned success.\n");
    wprintf(L"FSCTL_QUERY_FILE_SYSTEM_RECOGNITION thinks the filesystem is
\"%s\".\n", FsRi.FileSystem);

    if(hDisk != INVALID_HANDLE_VALUE)
    {
        CloseHandle(hDisk);
        hDisk = INVALID_HANDLE_VALUE;
    }

    return hr;
}

int wmain(int argc, WCHAR **argv)
{

      Check(L"\\\\.\\C:");
      return 0;
}
```

Take note that the FSCTL_QUERY_FILE_SYSTEM_RECOGNITION cannot be found in the Winioctl.h header file at the moment this program example was tested. It is intended for vendors and the minimum system to run this program is Windows 7/Server 2008 R2.

Naming a Volume

A label is a user-friendly name that is assigned to a volume, usually by an end user, to make it easier to recognize. A volume can have a label, a drive letter, both, or neither. To set the label for a volume, use the SetVolumeLabel() function. Several factors can make it difficult to identify specific volumes using only drive letters and labels.

1. One is that a **volume is not required to have a drive letter or a label**.
2. Another is that two different volumes can have the same label, which makes them indistinguishable except by drive letter.
3. A third factor is that drive letter assignments can change as volumes are added to and removed from the computer.

To solve this problem, the operating system uses volume **GUID paths** to identify volumes. These are strings of this form:

"\\?\Volume{*GUID*}\"

Where *GUID* is a globally unique identifier (GUID) that identifies the volume. A volume GUID path is sometimes referred to as a unique volume name, because a volume GUID path can refer only to one volume. However, this term is misleading, because a volume can have more than one volume GUID path.
The \\?\ prefix disables path parsing and is not considered to be part of the path. You must specify full paths when using volume GUID paths with the \\?\ prefix.
A **mounted folder** is an association between a folder on one volume and another volume, so that the folder path can be used to access the volume. For example, if you use the SetVolumeMountPoint()

function to create a mounted folder that associates the volume D:\ with the folder C:\MountD\, you can then use either path (D:\ or C:\MountD\) to access the volume D:\. A **volume mount point** is any user-mode path that can be used to access a volume. There are three types of volume mount points:

1. A drive letter, for example, C:\.
2. A volume GUID path, for example, \\?\Volume{26a21bda-a627-11d7-9931-806e6f6e6963}\.
3. A mounted folder, for example, C:\MountD\.

All volume and mounted folder functions that take a volume GUID path as an input parameter require the trailing backslash. All volume and mounted folder functions that return a volume GUID path provide the trailing backslash, but this is not the case with the CreateFile() function. You can open a volume by calling CreateFile() and omit the trailing backslash from the volume name you specify. CreateFile() processes a volume GUID path with an appended backslash as the root directory of the volume.
The operating system assigns a volume GUID path to a volume when the volume is first installed and when the volume is formatted. The volume and mounted folder functions use volume GUID paths to access volumes. To obtain the volume GUID path for a volume, use the GetVolumeNameForVolumeMountPoint() function.
Path lengths may be a concern when a mounted folder is created that associates a volume that has a deep directory tree with a directory on another volume. This is because the path of the volume is concatenated to the path of the directory. The globally defined constant MAX_PATH defines the maximum number of characters a path can have. You can avoid this constraint by doing either of the following:

1. Refer to volumes by their volume GUID paths.
2. Use the Unicode (W) versions of file functions, which support the \\?\ prefix.

Enumerating Volumes

To make a complete list of the volumes on a computer, or to manipulate each volume in turn, you can enumerate volumes. Within a volume, you can scan for mounted folders or other objects on the volume. Three functions allow an application to enumerate volumes on a computer:

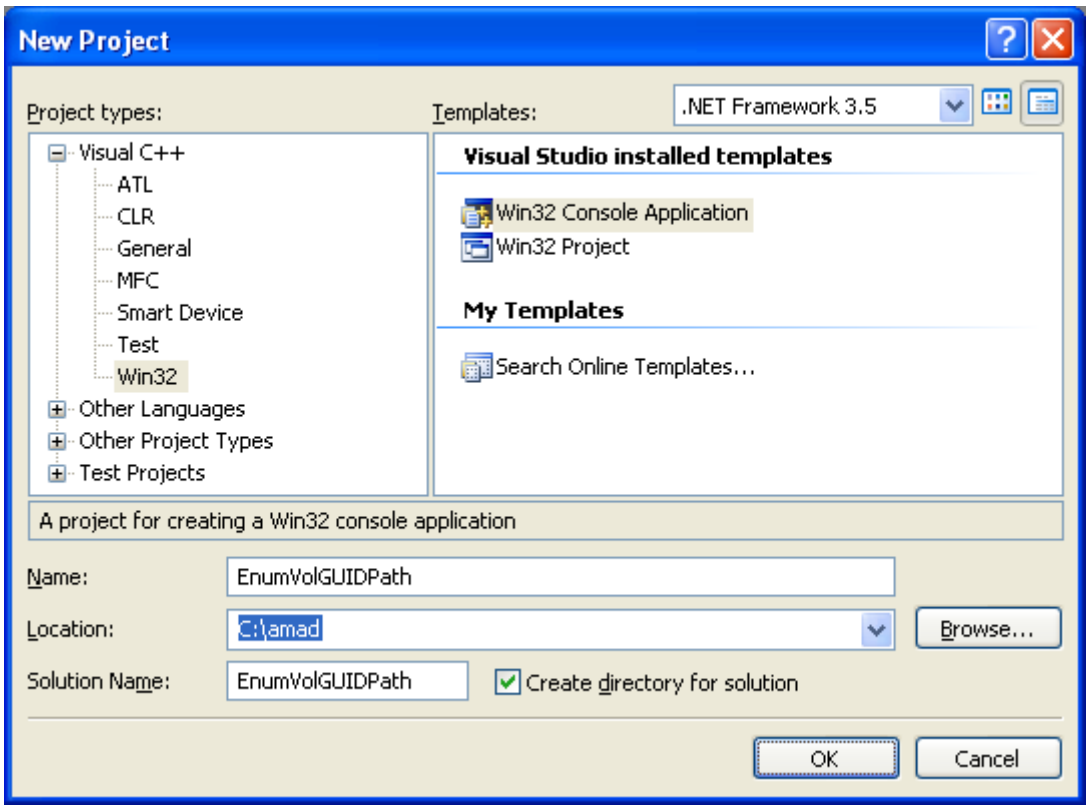
1. FindFirstVolume()
2. FindNextVolume()
3. FindVolumeClose()

These three functions operate in a manner very similar to the FindFirstFile(), FindNextFile(), and FindClose() functions.
Begin a search for volumes with FindFirstVolume(). If the search is successful, process the results according to the design of your application. Then use FindNextVolume() in a loop to locate and process each subsequent volume. When the supply of volumes is exhausted, close the search with FindVolumeClose().

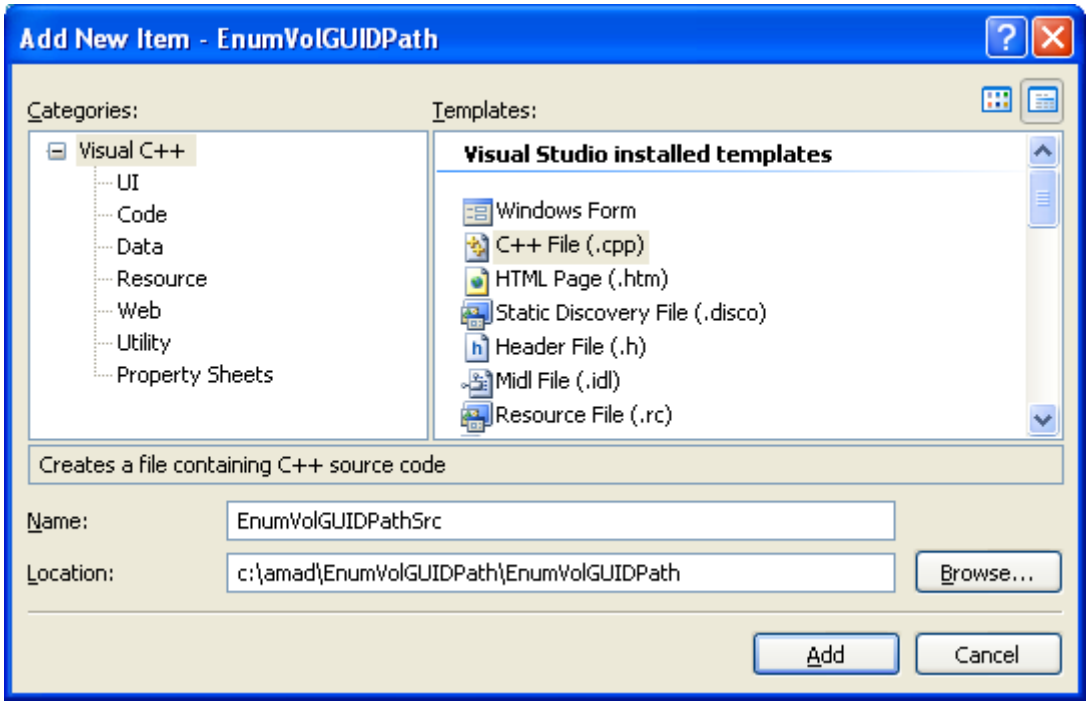
You should not assume any correlation between the order of the volumes that are returned by these functions and the order of the volumes that are returned by other functions or tools. In particular, do not assume any correlation between volume order and drive letters as assigned by the BIOS (if any) or the Disk Administrator.

Enumerating Volume GUID Paths Example

The following code example shows you how to obtain a volume GUID path for each volume associated with a drive letter that is currently in use on the computer. The code example uses the GetVolumeNameForVolumeMountPoint() function.
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.

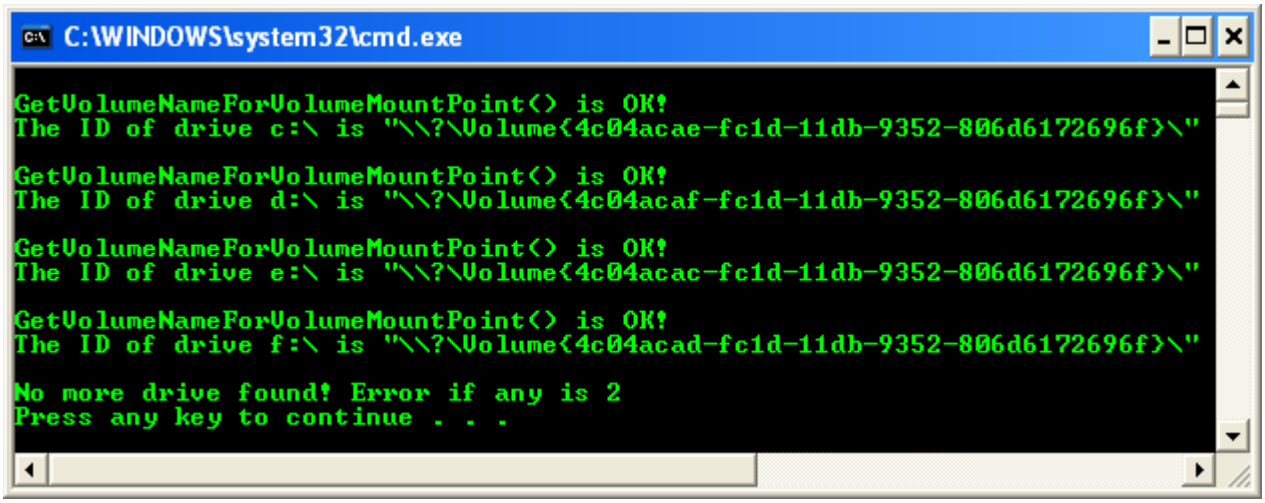


Add the following source code.

```
// Compile As C++ Code (/TP) and using Unicode character set
#include <windows.h>
#include <stdio.h>

#define BUFSIZE MAX_PATH

int main(void)
{
   BOOL bFlag;
   WCHAR Buf[BUFSIZE];         // temporary buffer for volume name
   WCHAR Drive[] = L"c:\\";    // template drive specifier
   WCHAR I;                    // generic loop counter

   // Walk through legal drive letters, skipping floppies.
   for (I = 'c'; I < 'z';  I++ )
   {
      // Stamp the drive for the appropriate letter.
      Drive[0] = I;

      bFlag = GetVolumeNameForVolumeMountPoint(
                 Drive,  // input volume mount point or directory
                 Buf,    // output volume name buffer
                 BUFSIZE // size of volume name buffer
              );

        // Should be non-zero
      if (bFlag != 0)
      {
              wprintf(L"\nGetVolumeNameForVolumeMountPoint() is OK!\n");
```

```
                wprintf(L"The ID of drive %s is \"%s\"\n", Drive, Buf);
        }
          // If zero
          else
          {
                wprintf(L"\nNo more drive found! Error if any is %d\n",
GetLastError());
                // ERROR_FILE_NOT_FOUND = 2 (0x2) - The system cannot find the
file specified. No more files
                break;
          }
    }
}
```

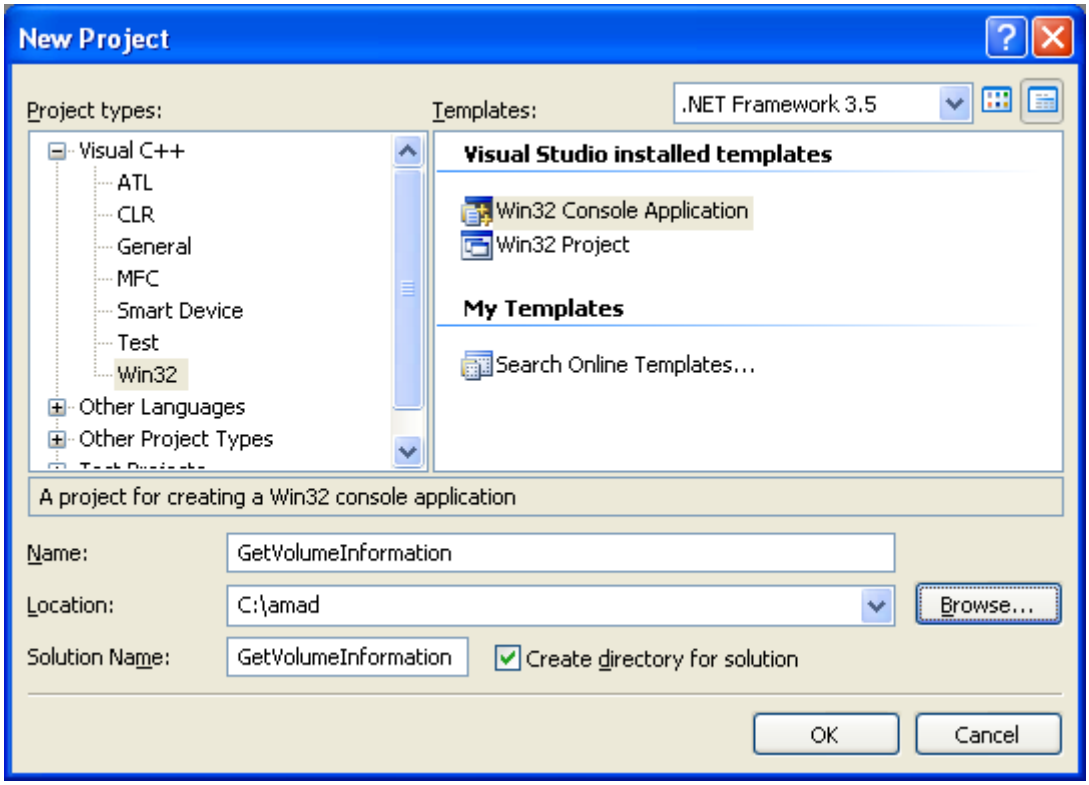
Build and run the project. The following screenshot shows a sample output. The e: and f: are thumb drives.



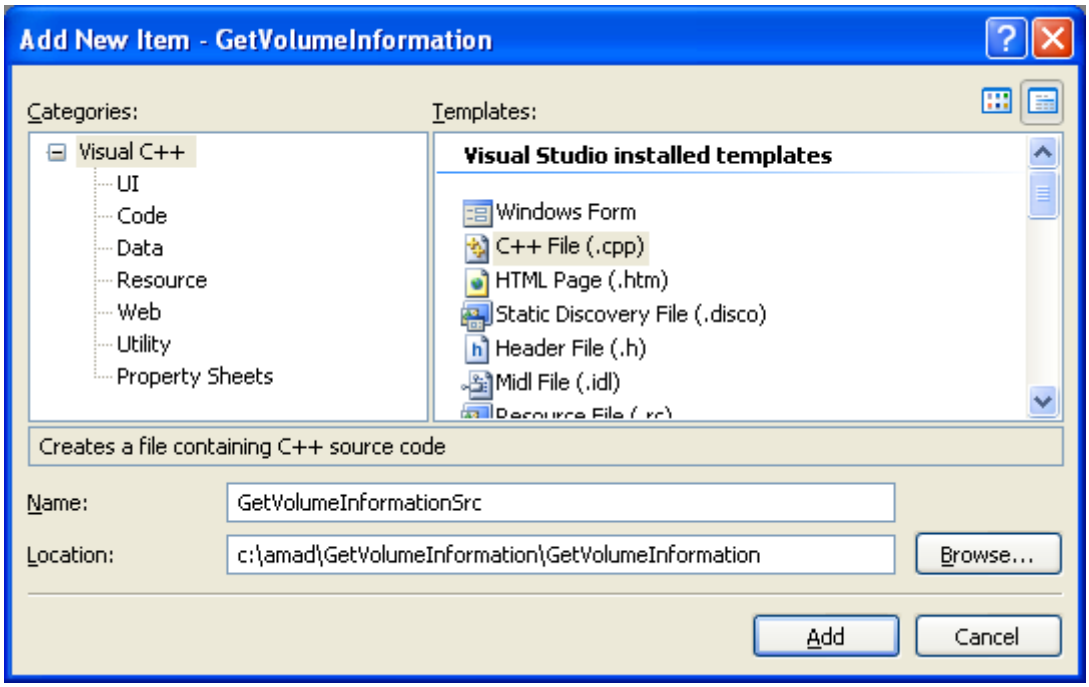
Obtaining Volume Information Program Example

The GetVolumeInformation() function retrieves information about the file system on a given volume. This information includes the volume name, volume serial number, file system name, file system flags, maximum length of a file name, and so on. Before you access files and directories on a given volume, you should determine the capabilities of the file system by using the GetVolumeInformation() function. This function returns values that you can use to adapt your application to work effectively with the file system.
In general, you should avoid using static buffers for file names and paths. Instead, use the values returned by GetVolumeInformation() to allocate buffers as you need them. If you must use static buffers, reserve 256 characters for file names and 260 characters for paths.
The GetSystemDirectory() and GetWindowsDirectory() functions retrieve the paths to the system directory and the Windows directory, respectively.
The GetDiskFreeSpace() function retrieves organizational information about a volume, including the number of bytes per sector, the number of sectors per cluster, the number of free clusters, and the total number of clusters. However, GetDiskFreeSpace() cannot report volume sizes that are greater

than 2 GB. To ensure that your application works with large capacity hard drives, use the GetDiskFreeSpaceEx() function.
The GetDriveType() function indicates whether the volume referenced by the specified drive letter is a removable, fixed, CD-ROM, RAM, or network drive.
The GetLogicalDrives() function identifies the volumes present. The GetLogicalDriveStrings() function retrieves a null-terminated string for each volume present. Use these strings whenever a root directory is required.
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.

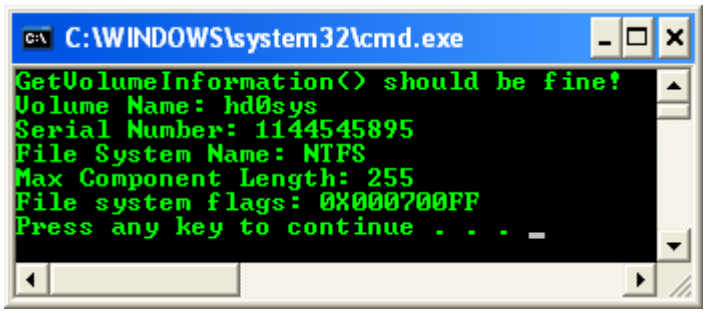


Add the following source code.

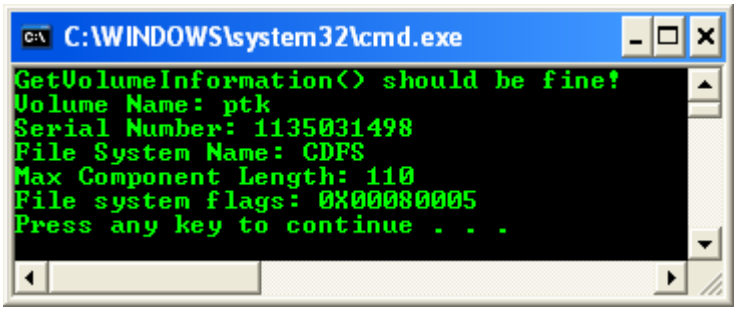
```
#include <stdio.h>
#include <Windows.h>

int wmain()
{
	// + 1 is for NULL
	WCHAR volumeName[MAX_PATH + 1] = { 0 };
	WCHAR fileSystemName[MAX_PATH + 1] = { 0 };
	DWORD serialNumber = 0;
	DWORD maxComponentLen = 0;
	DWORD fileSystemFlags = 0;

	if (GetVolumeInformation(
		L"C:\\", // L"\\MyServer\MyShare\"
	  volumeName,
	  sizeof(volumeName),
	  &serialNumber,
	  &maxComponentLen,
	  &fileSystemFlags,
	  fileSystemName,
	  sizeof(fileSystemName)) == TRUE)

	{
		wprintf(L"GetVolumeInformation() should be fine!\n");
		wprintf(L"Volume Name: %s\n", volumeName);
		wprintf(L"Serial Number: %lu\n", serialNumber);
		wprintf(L"File System Name: %s\n", fileSystemName);
		wprintf(L"Max Component Length: %lu\n", maxComponentLen);
		wprintf(L"File system flags: 0X%.08X\n", fileSystemFlags);
	}
```

```
	else
	{
		wprintf(L"GetVolumeInformation() failed, error %u\n",
GetLastError());
	}

	return 0;
}
```

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

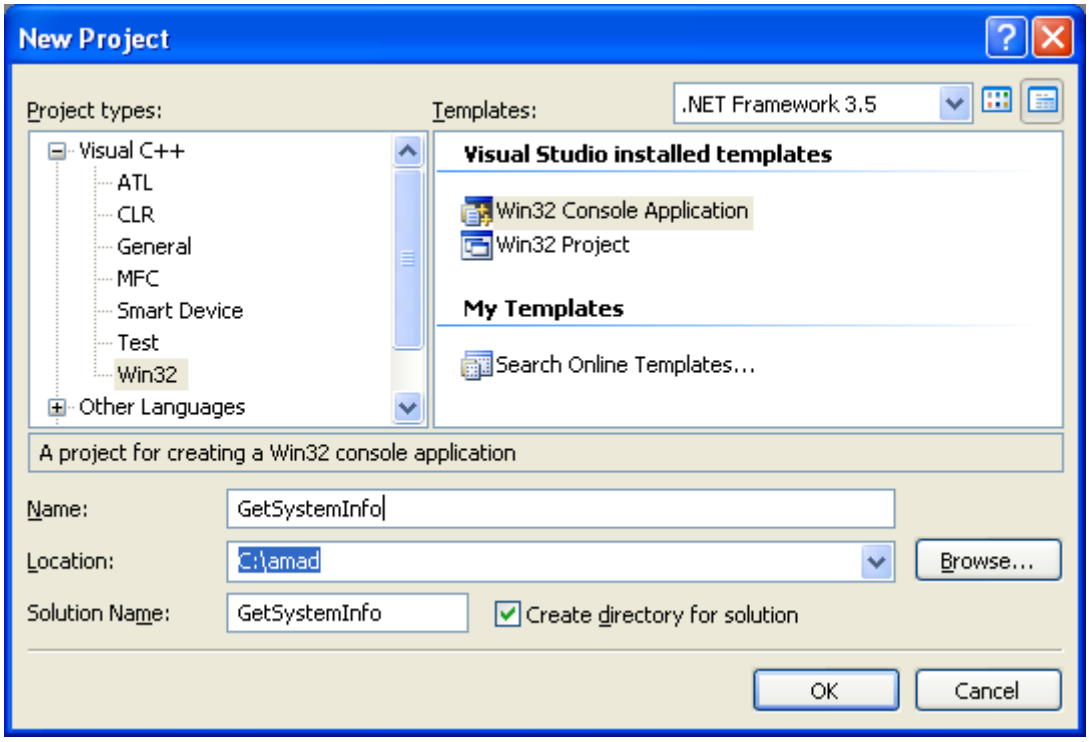


The following sample output is querying the DVD/CD drive with CD inside the drive.

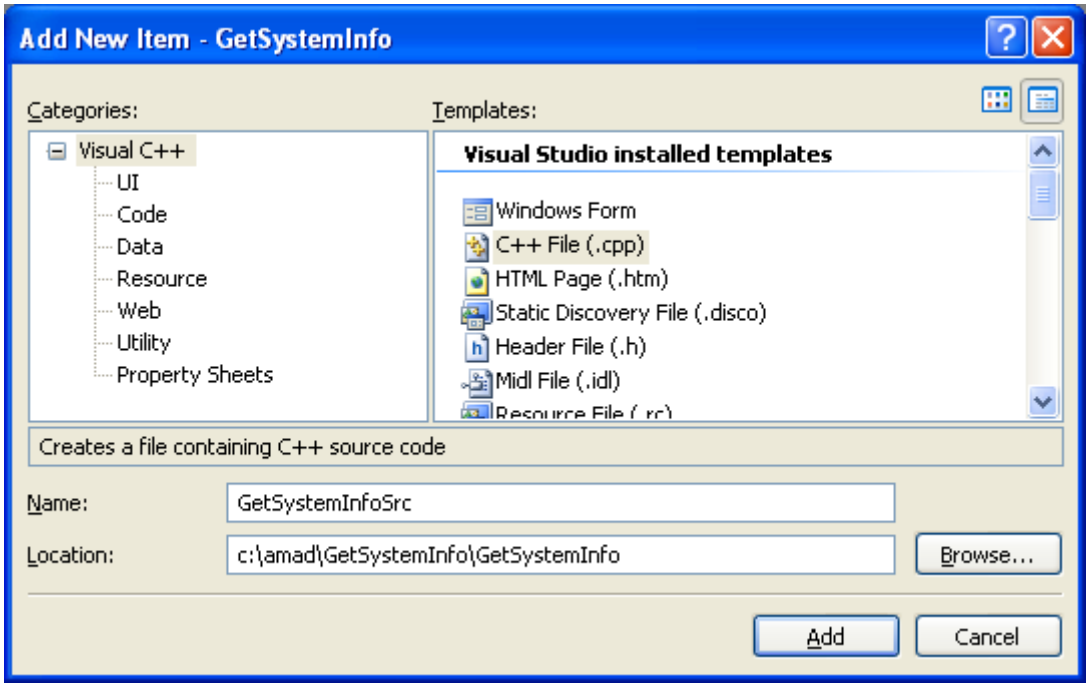


Getting the System Information Program Example

Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



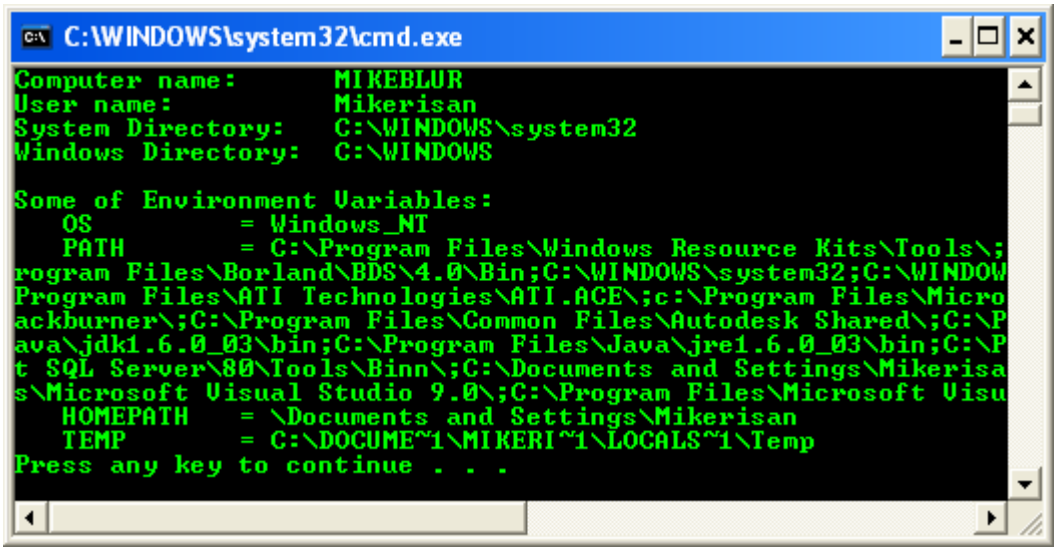
Add the following source code.

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

```
WCHAR* envVarStrings[] =
{
  L"OS          = %OS%",
  L"PATH        = %PATH%",
  L"HOMEPATH    = %HOMEPATH%",
  L"TEMP        = %TEMP%"
};
#define  ENV_VAR_STRING_COUNT  (sizeof(envVarStrings)/sizeof(WCHAR*))
#define INFO_BUFFER_SIZE 32767

void printError( WCHAR* msg );

int wmain( )
{
  DWORD i;
  WCHAR  infoBuf[INFO_BUFFER_SIZE];
  DWORD  bufCharCount = INFO_BUFFER_SIZE;

  // Get and display the name of the computer.
  bufCharCount = INFO_BUFFER_SIZE;
  if(!GetComputerName( infoBuf, &bufCharCount))
    printError(L"GetComputerName()");
  wprintf(L"\nComputer name:      %s", infoBuf);

  // Get and display the user name.
  bufCharCount = INFO_BUFFER_SIZE;
  if(!GetUserName(infoBuf, &bufCharCount))
    printError(L"GetUserName()");
  wprintf(L"\nUser name:          %s", infoBuf);

  // Get and display the system directory.
  if(!GetSystemDirectory(infoBuf, INFO_BUFFER_SIZE))
    printError(L"GetSystemDirectory()");
  wprintf(L"\nSystem Directory:   %s", infoBuf);

  // Get and display the Windows directory.
  if(!GetWindowsDirectory(infoBuf, INFO_BUFFER_SIZE))
    printError(L"GetWindowsDirectory()");
  wprintf(L"\nWindows Directory:  %s", infoBuf);

  // Expand and display a few environment variables.
  wprintf(L"\n\nSome of Environment Variables:");
  for(i = 0; i < ENV_VAR_STRING_COUNT; ++i)
  {
    bufCharCount = ExpandEnvironmentStrings(envVarStrings[i],
infoBuf,INFO_BUFFER_SIZE);
    if( bufCharCount > INFO_BUFFER_SIZE )
      wprintf(L"\n\t(Buffer too small to expand: \"%s\")",envVarStrings[i]);
    else if(!bufCharCount)
      printError(L"ExpandEnvironmentStrings()");
    else
      wprintf(L"\n   %s", infoBuf);
  }
  wprintf(L"\n");
```

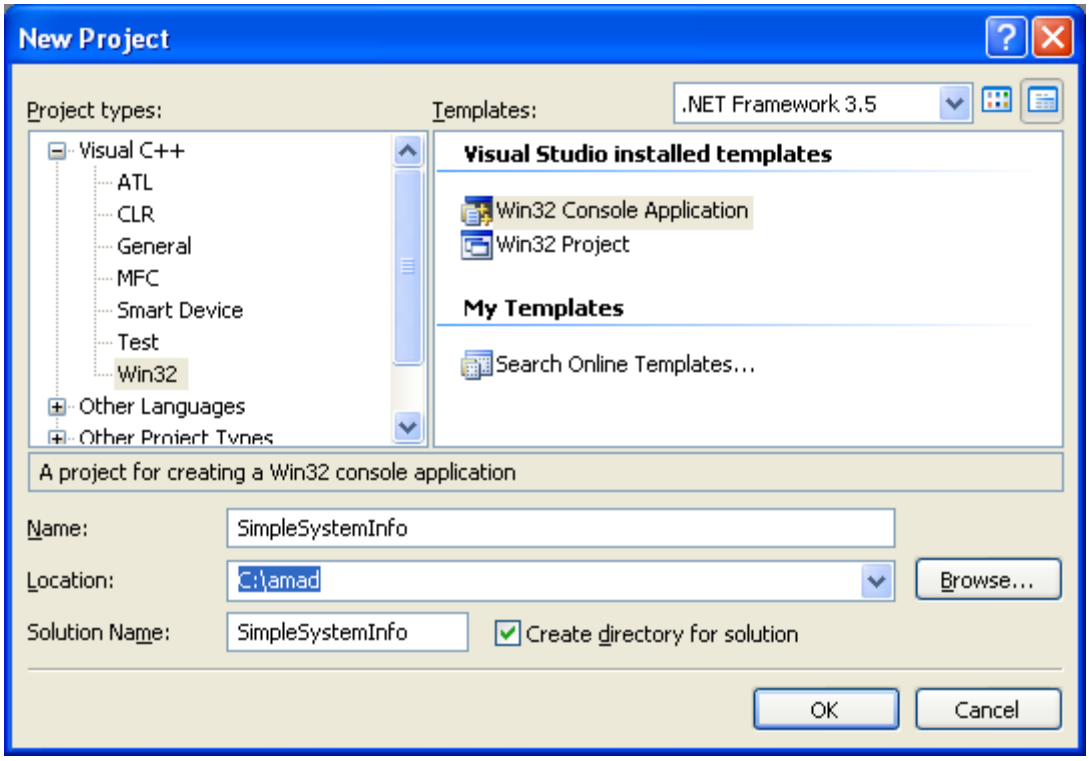
```
  return 0;
}

void printError(WCHAR* msg)
{
  DWORD eNum;
  WCHAR sysMsg[256];
  WCHAR* p;

  eNum = GetLastError();
  FormatMessage( FORMAT_MESSAGE_FROM_SYSTEM |
         FORMAT_MESSAGE_IGNORE_INSERTS,
         NULL, eNum,
         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
         sysMsg, 256, NULL);

  // Trim the end of the line and terminate it with a null
  p = sysMsg;
  while(( *p > 31) || (*p == 9))
    ++p;
  do
  {
        *p-- = 0;
  } while((p >= sysMsg) && (( *p == '.') || (*p < 33)));

  // Display the message
  wprintf(L"\n\t%s failed with error %d (%s)", msg, eNum, sysMsg);
}
```

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

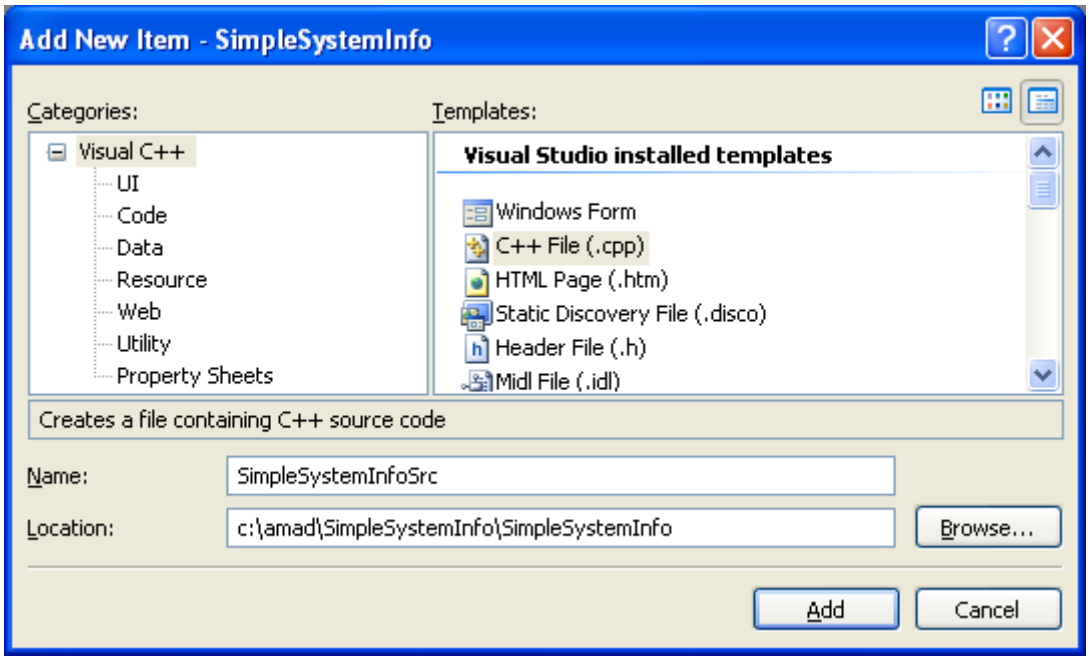


Another Basic Windows System Information Program Example

Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

```
// Extracting some computer information program example
#include<windows.h>
#include<stdio.h>
```

```
#define TOTALBYTES    2048

void PrintSystemName()
{
	WCHAR compName[40];
	DWORD size = 40;

	if(GetComputerName(compName,&size)!=0)
		wprintf(L"Computer Name : %s ",compName);
}

void PrintUserName()
{
	WCHAR uName[40];
	DWORD size = 40;

	if(GetUserName(uName,&size) != 0)
		wprintf(L"\nLogged user name : %s",uName);
}

void PrintSystemUpTime()
{
	unsigned int t,d,h,m,s;

	// Retrieves the number of milliseconds that have elapsed since
	// the system was started, up to 49.7 days.
	t = GetTickCount();

	// Get the seconds
	t/=1000;
	//days
	d = t/86400;
	t = t%86400;
	// hours
	h = t/3600;
	t = t%3600;
	// minutes
	m = t/60;
	t = t%60;
	// seconds
	s = t;

	wprintf(L"\nSystem Up Time(D:H:M:S) = %u:%u:%u:%u",d,h,m,s);
}
void PrintDrivesInfo()
{
	int drives,i=0;
	__int64 nFree, nTotal, nHDFree=0, nHDTotal=0;
	WCHAR dName[40],volName[40];

	wprintf(L"\n\nHard Disk Info:");
	wprintf(L"\n---------------");

	// Returns the drive bitmasks
	drives = GetLogicalDrives();
```

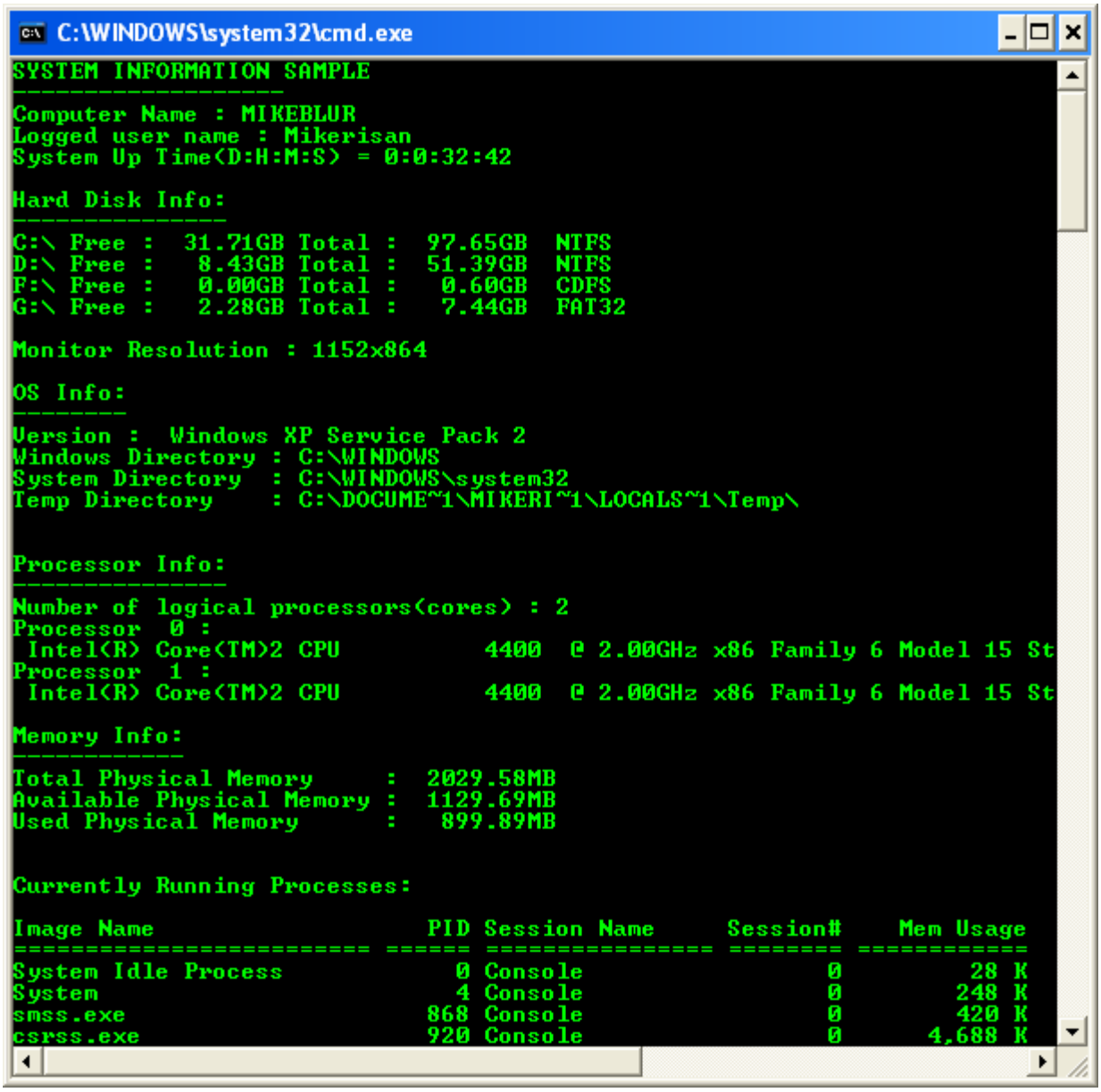
```
	// Iterate all the available drives
	while(drives != 0)
	{
		// Do the logical AND
		if((drives&1) == 1)
		{
			// Iterate starting from drive A
			wsprintf(dName,L"%c:\\",'A'+i);

			// Retrieves information about the amount of space that is
available on a
			// disk volume, which is the total amount of space, the total
amount of
			// free space, and the total amount of free space available to
the user
			// that is associated with the calling thread.

	if(GetDiskFreeSpaceEx(dName,(PULARGE_INTEGER)&nFree,(PULARGE_INTEGER)&nTot
al,NULL)!= 0)
			{
				wprintf(L"\n%s",dName);
				nHDFree += nFree;
				nHDTotal += nTotal;
				wprintf(L" Free : %6.2I64fGB Total : %6.2I64fGB  ",
nFree/(1024*1024*1024.0),nTotal/(1024*1024*1024.0));

				// Retrieves information about the file system and
volume
				// associated with the specified root directory.
				if(GetVolumeInformation(
					dName,
					NULL,
					0,
					NULL,
					NULL,
					NULL,
					volName,
					sizeof(volName)) !=0)

					wprintf(L"%s",volName);
			}
		}
		drives>>=1;
		i++;
	}

	// Total storage and free, uncomment to see it
	// wprintf(L"\n==========================================");
	// wprintf(L"\n    Free : %6.2I64fGB Total :
%6.2I64fGB",nHDFree/(1024*1024*1024.0),nHDTotal/(1024*1024*1024.0));
	// wprintf(L"\n==========================================");
}

void PrintMonitorResolution()
{
```

```
	int width,height;

	// GetSystemMetrics() function retrieves the dimensions — widths and
heights — of
	// Windows display elements and system configuration settings in pixels
	width = GetSystemMetrics(SM_CXSCREEN);
	height = GetSystemMetrics(SM_CYSCREEN);
	wprintf(L"\n\nMonitor Resolution : %dx%d",width,height);
}

void PrintOSInfo()
{
	WCHAR windirName[MAX_PATH];
	OSVERSIONINFO verInfo = {sizeof(OSVERSIONINFO)};
	wprintf(L"\n\nOS Info: ");
	wprintf(L"\n--------");
	wprintf(L"\nVersion : ");

	// Retrieves information about the current operating system.
	GetVersionEx(&verInfo);
	if(verInfo.dwMajorVersion == 4 && verInfo.dwMinorVersion == 10)
		wprintf(L" Windows 98 %s",verInfo.szCSDVersion);
	if(verInfo.dwMajorVersion == 5 && verInfo.dwMinorVersion == 0)
		wprintf(L" Windows 2000 %s",verInfo.szCSDVersion);
	if(verInfo.dwMajorVersion == 5 && verInfo.dwMinorVersion == 1)
		wprintf(L" Windows XP %s",verInfo.szCSDVersion);
	if(verInfo.dwMajorVersion == 5 && verInfo.dwMinorVersion == 2)
		wprintf(L" Windows 2003 %s",verInfo.szCSDVersion);

	// Get Windows directory
	GetWindowsDirectory(windirName,55);
	wprintf(L"\nWindows Directory : %s ",windirName);

	// Get Windows system directory
	GetSystemDirectory(windirName,55);
	wprintf(L"\nSystem Directory  : %s ",windirName);

	// Get the Windows temporary directory
	GetTempPath(MAX_PATH,windirName);
	wprintf(L"\nTemp Directory    : %s \n",windirName);
}

void PrintProcessorInfo()
{
	// Pointer to
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor
	HKEY hKey,tempKey;
	WCHAR subKeyName[40];
	DWORD BufferSize = TOTALBYTES;
	LPBYTE valBuf = (LPBYTE)malloc(BufferSize);
	int i=0,t;
	DWORD size=100;

	wprintf(L"\n\nProcessor Info:");
	wprintf(L"\n---------------");
```

```
	// Embedding the assembly in C/C++ code
	__asm
	{
		// 01h for getting number of core present in the physical processor
		mov eax,01h
		cpuid
		mov t,ebx
	}
	wprintf(L"\nNumber of logical processors(cores) : %d",(t>>16)&0xff);

	// Read the processor from registry
	// Open the registry key
	if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,
		L"HARDWARE\\DESCRIPTION\\System\\CentralProcessor",
		0,
		KEY_READ,
		&hKey)== ERROR_SUCCESS)
	{
		// Enumerate the keys
		while(RegEnumKey(hKey,i++,subKeyName,40) != ERROR_NO_MORE_ITEMS)
		{
			if(RegOpenKeyEx(hKey,subKeyName,0,KEY_READ,&tempKey) ==
ERROR_SUCCESS)
			{
				size = 100;

				// Retrieves the type and data for the
ProcessorNameString.
				if(RegQueryValueEx(tempKey,
					L"ProcessorNameString",
					NULL,
					NULL,
					valBuf,
					&size) == ERROR_SUCCESS)

					wprintf(L"\nProcessor  %s :\n
%s",subKeyName,valBuf);

				size = 100;

				// Retrieves the type and data for the Identifier.
				if(RegQueryValueEx(
					tempKey,
					L"Identifier",
					NULL,
					NULL,
					valBuf,
					&size) == ERROR_SUCCESS)

					wprintf(L" %s",valBuf);
				RegCloseKey(tempKey);
			}
		}
		RegCloseKey(hKey);
	}
	free(valBuf);
```

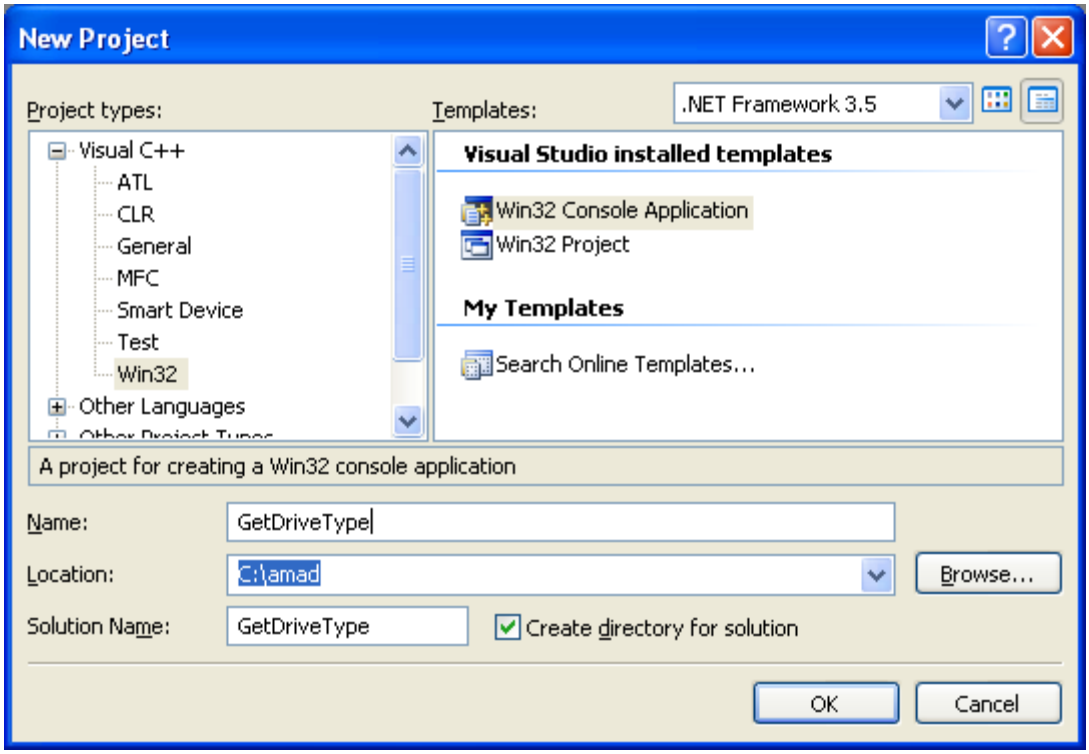
```
}

void PrintMemoryInfo()
{
	MEMORYSTATUSEX ms={sizeof(MEMORYSTATUSEX)};

	// Retrieves information about the system's current
	// usage of both physical and virtual memory.
	GlobalMemoryStatusEx(&ms);

	wprintf(L"\n\nMemory Info:");
	wprintf(L"\n------------");
	wprintf(L"\nTotal Physical Memory     : %8.2I64fMB \nAvailable Physical
Memory : %8.2I64fMB \nUsed Physical Memory      : %8.2I64fMB
\n\n",ms.ullTotalPhys/(1024*1024.0),ms.ullAvailPhys/(1024*1024.0),ms.ullTotalPhy
s/(1024*1024.0)-ms.ullAvailPhys/(1024*1024.0));
}

int wmain(int argc, WCHAR **argv)
{
	wprintf(L"SYSTEM INFORMATION SAMPLE\n-------------------\n");
	PrintSystemName();
	PrintUserName();
	PrintSystemUpTime();
	PrintDrivesInfo();
	PrintMonitorResolution();
	PrintOSInfo();
	PrintProcessorInfo();
	PrintMemoryInfo();

	wprintf(L"\nCurrently Running Processes:\n----------------------------");
	// Just ask tasklist command
	system("tasklist");
}
```

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

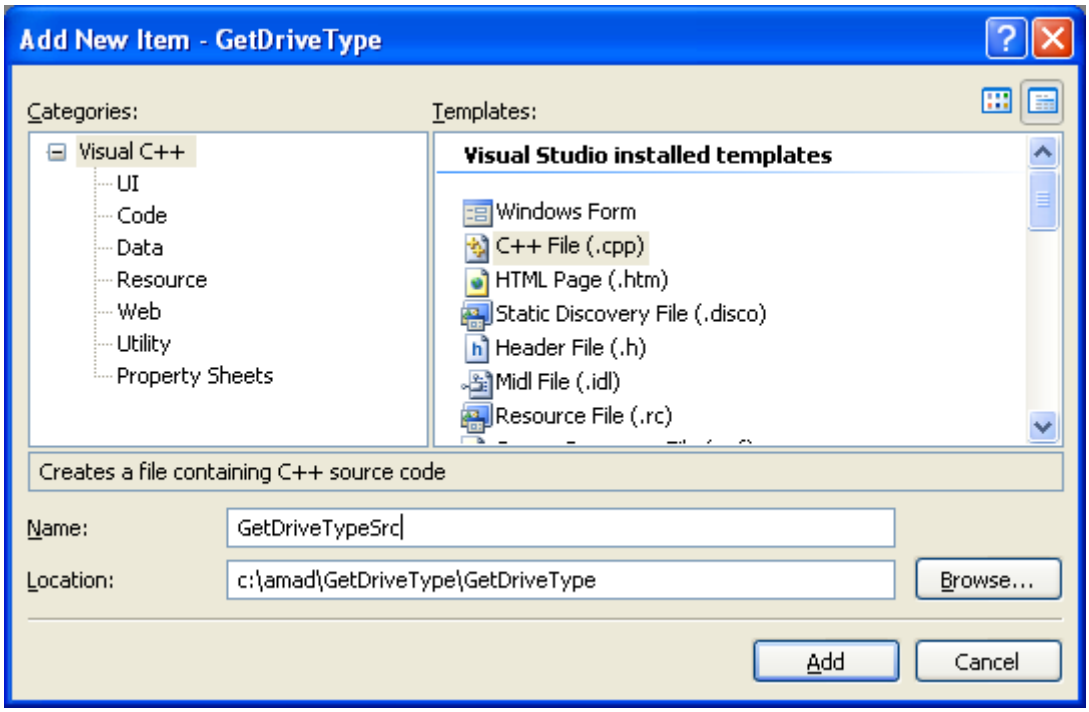


Getting Logical Drive Program Example

Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



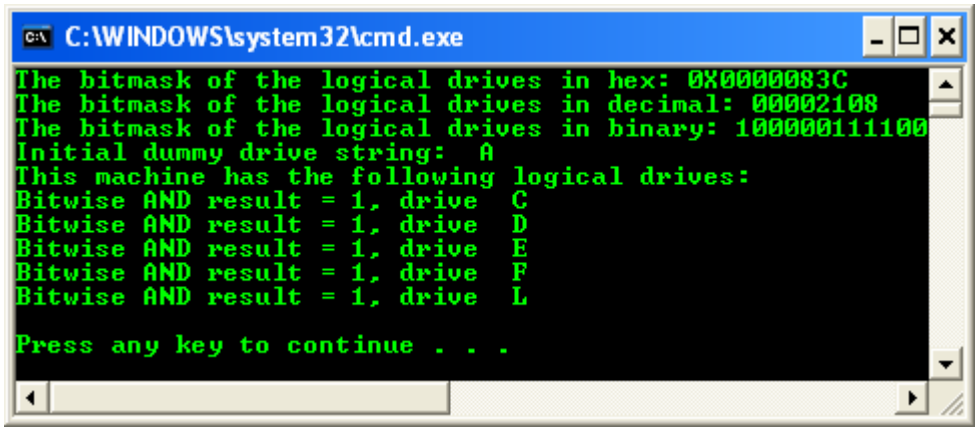
Add the following source code.

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

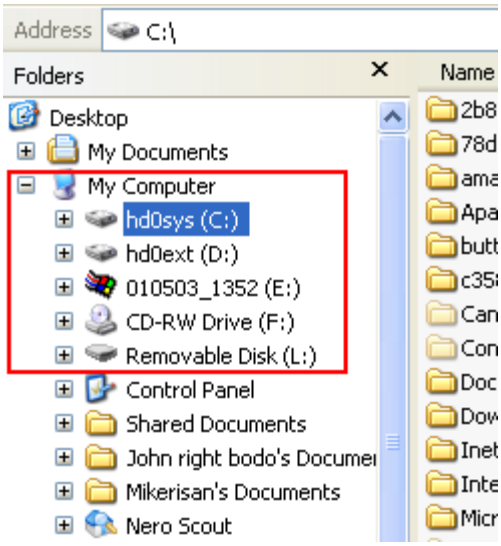
```
#include <stdio.h>

// Optional function: Decimal to binary
void decnumtobin(DWORD deci)
{
	DWORD input = deci;
	DWORD i;
	DWORD count = 0;
	DWORD binary[128];

	do
	{
		// Modulus 2 to get 1 or a 0
		i = input%2;
		// Load elements into the binary array
		binary[count] = i;
		// Divide input by 2 for binary decrement
		input = input/2;
		// Count the binary digits
		count++;
	}while (input > 0);

	// Reverse and output binary digits
	wprintf(L"The bitmask of the logical drives in binary: ");
	do
	{
		wprintf(L"%d", binary[count - 1]);
		count--;
	} while (count > 0);
	wprintf(L"\n");
}

int main()
{
	// Must give initial value and then let the
	// while loop iterates
	// There is wierd thing here, we need to provide a space
	// at the beginning...what wrong here?
	WCHAR szDrive[] = L" A";
	// Get the logical drive mask
	DWORD uDriveMask = GetLogicalDrives();

	// Display the drive mask
	wprintf(L"The bitmask of the logical drives in hex: 0X%.8X\n",
uDriveMask);
	wprintf(L"The bitmask of the logical drives in decimal: %.8d\n",
uDriveMask);
	decnumtobin(uDriveMask);
	wprintf(L"Initial dummy drive string: %s\n", szDrive);

	// Verify the returned drive mask
	if(uDriveMask == 0)
		wprintf(L"GetLogicalDrives() failed with error code: %d\n",
GetLastError());
	else
	{
```

```
            wprintf(L"This machine has the following logical drives:\n");
            while(uDriveMask)
            {
                  // use the bitwise AND, 1-available (1 & 1), 0-not available
                  // the binary representation is in reverse lol
                  if(uDriveMask & 1)
                  {
                        // Just print out the available drives
                        wprintf(L"Bitwise AND result = %u, drive %s\n",
(uDriveMask & 1), szDrive);
                  }
                  // increment for next...
                  szDrive[1]++;
                  // shift the bitmask binary right
                  uDriveMask >>= 1;
            }
            wprintf(L"\n");
      }
      return 0;
}
```

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

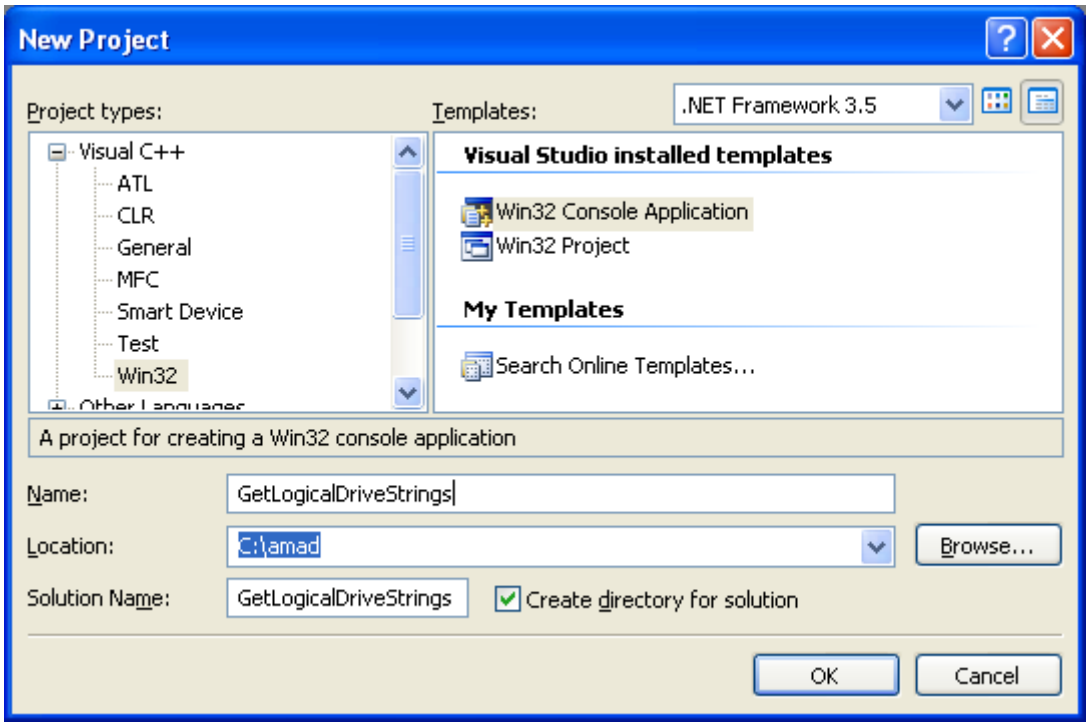


The following Figure shows the drives when seen in Windows explorer.

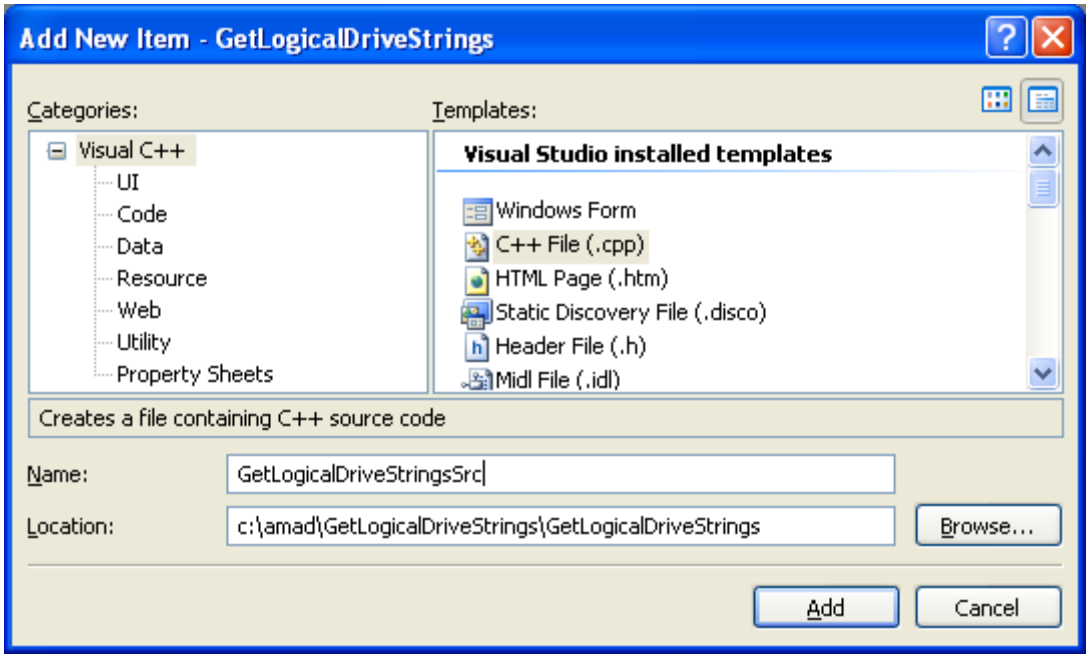


Getting the Logical Drive String Program Example

Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.

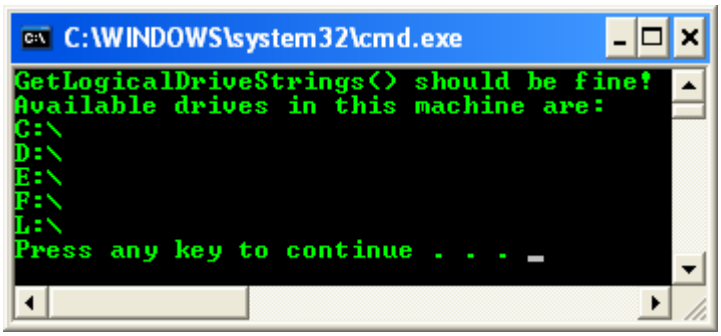


Add the following source code.

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

#define BUFSIZE 512

int wmain(int argc, WCHAR *argv[])
{
	// Translate path with device name to drive letters.
	WCHAR szTemp[BUFSIZE];
	// Hmmm...why the first index need to be NULL?
	szTemp[0] = '\0';
	// Allocate extra space for NULL lol! An initial value
	// WCHAR szDrive[3] = L" :";
	WCHAR szDrive[5] = L" :\\";
	// Initially not found
	BOOL bFound = FALSE;
	// Point pointer p to the temporary buffer
	WCHAR *p = szTemp;

	// Fills szTemp buffer with strings that specify valid drives in the
system.
	if (GetLogicalDriveStrings(BUFSIZE-1, szTemp))
	{
		wprintf(L"GetLogicalDriveStrings() should be fine!\n");
		wprintf(L"Available drives in this machine are:\n");
		do
		{
			// Copy the drive letter to the template string
			// Both pointers point to the same data, *p will
			// be used to skip the NULL
			*szDrive = *p;
```

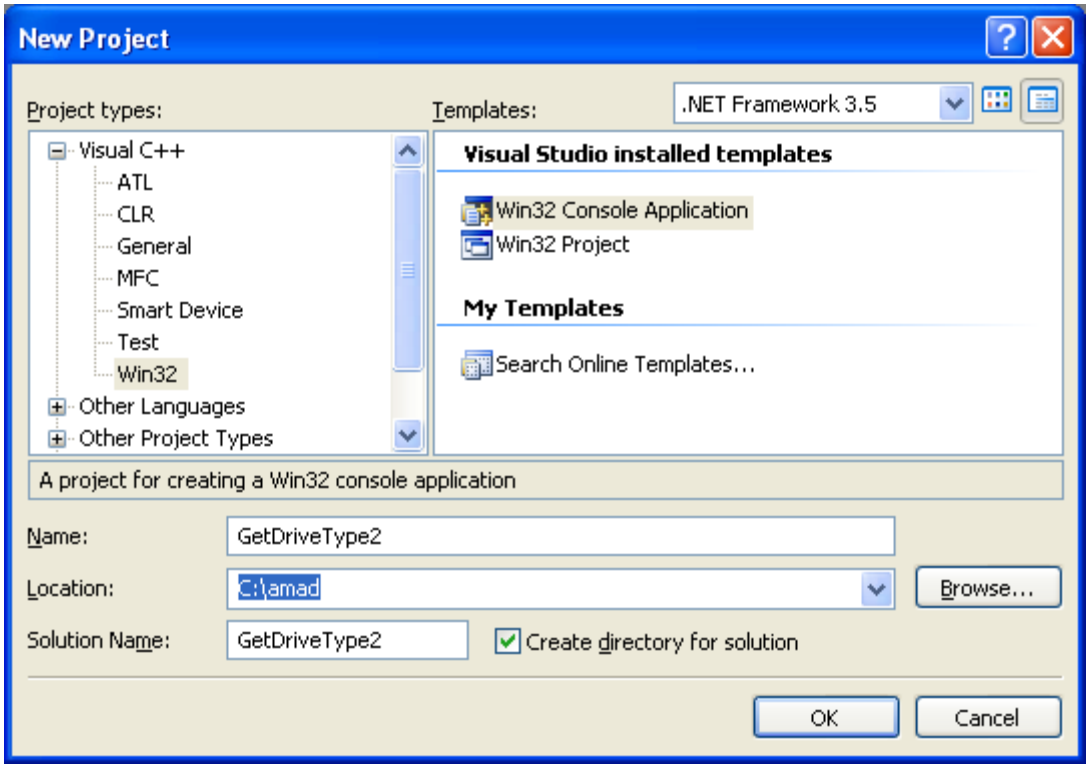
```
                    // Print the found drives
                    wprintf(L"%s\n", szDrive);
                    // This is WIERD!!! Can you see the while was used twice!!!
                    while (*p++);// skip the next NULL character, starts reading
new drive string
              } while (!bFound && *p); // end of string
       }
       else
              wprintf(L"GetLogicalDriveStrings() failed, error %u\n",
GetLastError());
       return 0;
}
```

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

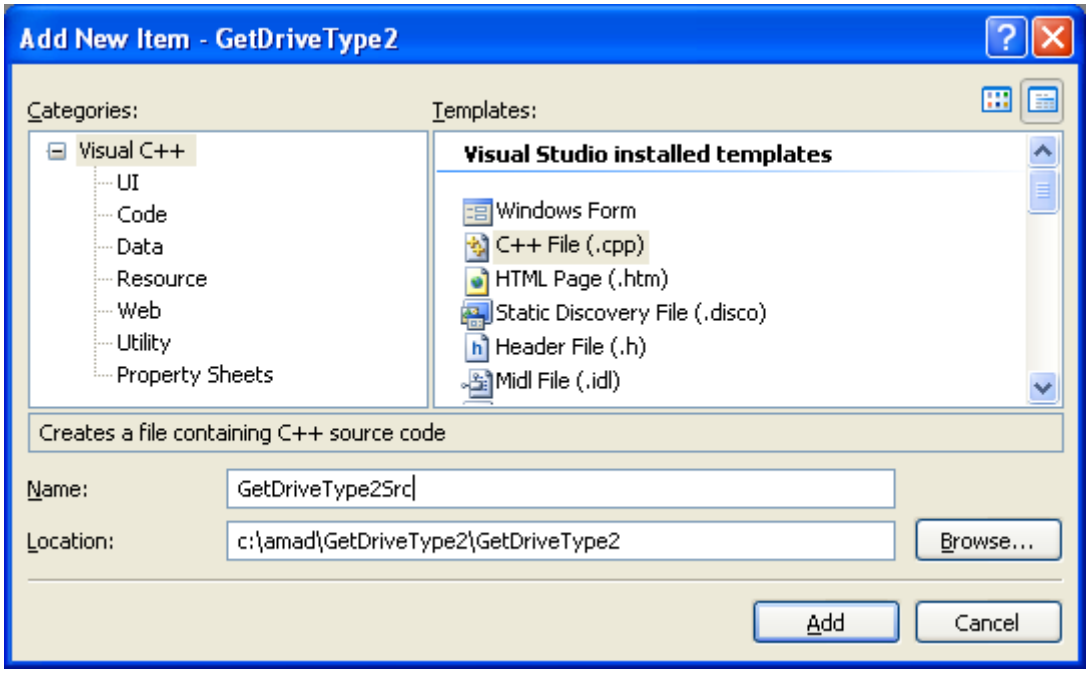


Getting Drive Type Program Example

Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



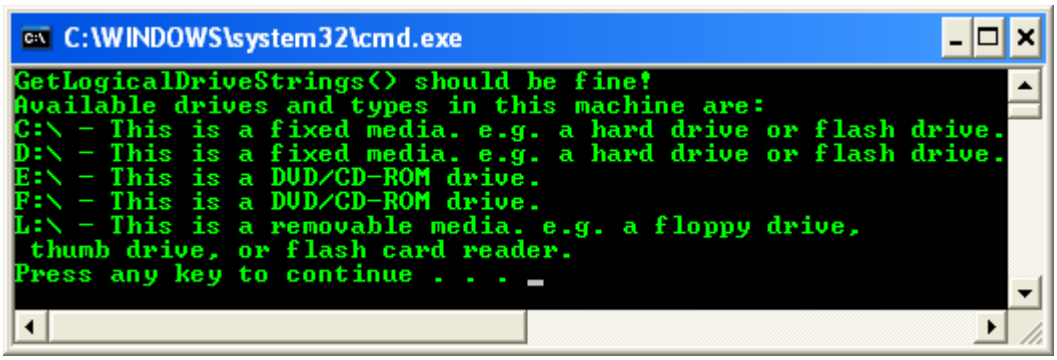
Add the following source code.

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

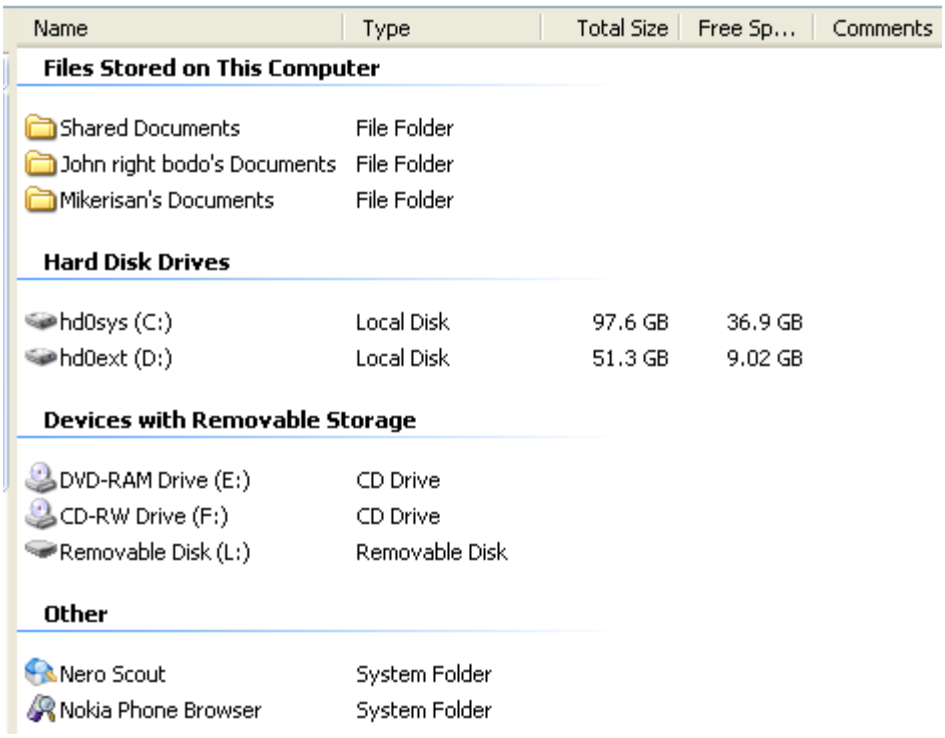
```
#define BUFSIZE 512

int wmain(int argc, WCHAR *argv[])
{
	// Translate path with device name to drive letters.
	WCHAR szTemp[BUFSIZE];
	// Hmmm...why the first index need to be NULL?
	szTemp[0] = '\0';
	// Allocate extra space for NULL lol! An initial value
	// WCHAR szDrive[2] = L" ";
	// WCHAR szDrive[3] = L" :";
	WCHAR szDrive[5] = L" :\\";
	// Initially not found
	BOOL bFound = FALSE;
	// Point pointer p to the temporary buffer
	WCHAR *p = szTemp;
	UINT uDriveRet;

	// Fills szTemp buffer with strings that specify valid drives in the
system.
	if (GetLogicalDriveStrings(BUFSIZE-1, szTemp))
	{
		wprintf(L"GetLogicalDriveStrings() should be fine!\n");
		wprintf(L"Available drives and types in this machine are:\n");
		do
		{
			// Copy the drive letter to the template string
			// Both pointers point to the same data, *p will
			// be used to skip the NULL
			*szDrive = *p;
			// Print the found drives
			wprintf(L"%s - ", szDrive);
			uDriveRet = GetDriveType(szDrive);

			switch(uDriveRet)
			{
			case DRIVE_UNKNOWN: wprintf(L"This drive type cannot be
determined.\n");
				break;
			case DRIVE_NO_ROOT_DIR: wprintf(L"The root path is invalid.
e.g. there is\n no volume mounted at the specified path.\n");
				break;
			case DRIVE_REMOVABLE: wprintf(L"This is a removable media.
e.g. a floppy drive,\n thumb drive, or flash card reader.\n");
				break;
			case DRIVE_FIXED: wprintf(L"This is a fixed media. e.g. a hard
drive or flash drive.\n");
				break;
			case DRIVE_REMOTE: wprintf(L"This is a remote (network)
drive.\n");
				break;
			case DRIVE_CDROM: wprintf(L"This is a DVD/CD-ROM drive.\n");
				break;
			case DRIVE_RAMDISK: wprintf(L"This is a RAM disk.\n");
				break;
```

```
                    default: wprintf(L"I don't know this error, %u\n",
GetLastError());
                    }

                    // This is WIERD!!! Can you see the while was used twice!!!
                    while (*p++);// skip the next NULL character, starts reading
new drive string
             } while (!bFound && *p); // end of string
      }
      else
             wprintf(L"GetLogicalDriveStrings() failed, error %u\n",
GetLastError());
      return 0;
}
```

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



The following Figure is the drives seen through the Windows explorer.



Change Journals

An automatic backup application is one example of a program that must check for changes to the state of a volume to perform its task. The brute force method of checking for changes in directories or files is to scan the entire volume. However, this is often not an acceptable approach because of the decrease in system performance it would cause. Another method is for the application to register a directory notification (by calling the FindFirstChangeNotification() or ReadDirectoryChangesW() functions) for the directories to be backed up. This is more efficient than the first method, however, it requires that an application be running at all times. Also, if a large number of directories and files must be backed up, the amount of processing and memory overhead for such an application might also cause the operating system's performance to decrease.
To avoid these disadvantages, the NTFS file system maintains a change journal. When any change is made to a file or directory in a volume, the change journal for that volume is updated with a description of the change and the name of the file or directory.
Change journals are also needed to recover file system indexing, for example after a computer or volume failure. The ability to recover indexing means the file system can avoid the time-consuming process of reindexing the entire volume in such cases.

Change Journal Records

As files, directories, and other NTFS file system objects are added, deleted, and modified, the NTFS file system enters change journal records in streams, one for each volume on the computer. Each record indicates the type of change and the object changed. The offset from the beginning of the stream for a particular record is called the update sequence number (USN) for the particular record. New records are appended to the end of the stream.

The NTFS file system may delete old records in order to conserve space. If needed records have been deleted, the indexing service recovers by re-indexing the volume, as it does when no change journal exists.
The change journal logs only the fact of a change to a file and the reason for the change (for example, write operations, truncation, lengthening, deletion, and so on). It does not record enough information to allow reversing the change.
In addition, multiple changes to the same file may result in only one reason flag being added to the current record. If the same kind of change occurs more than once, the NTFS file system does not write a new record for the changes after the first. For example, several write operations with no intervening close and reopen operations result in only one change record with the reason flag USN_REASON_DATA_OVERWRITE set. To illustrate how the change journal works, suppose a user accesses a file in the following order:

1. Writes to the file.
2. Sets the time stamp for the file.
3. Writes to the file.
4. Truncates the file.
5. Writes to the file.
6. Closes the file.

In this case, the NTFS file system takes the following actions in the change journal (where | indicates a bitwise OR operation).

Event	NTFS file system action
Initial write operation	The NTFS file system writes a new USN record with the USN_REASON_DATA_OVERWRITE reason flag set.
Setting of file time stamp	The NTFS file system writes a new USN record with the flag setting USN_REASON_DATA_OVERWRITE | USN_REASON_BASIC_INFO_CHANGE.
Second write operation	The NTFS file system does not write a new USN record. Because USN_REASON_DATA_OVERWRITE is already set for the existing record, no changes are made to the record.
File truncation	The NTFS file system writes a new USN record with the flag setting USN_REASON_DATA_OVERWRITE | USN_REASON_BASIC_INFO_CHANGE | USN_REASON_DATA_TRUNCATION.
Third write operation	The NTFS file system does not write a new USN record. Because USN_REASON_DATA_OVERWRITE is already set for the existing record, no changes are made to the record.
Close operation	If the user making changes is the only user of the file, the NTFS file system writes a new USN record with the following flag setting: USN_REASON_DATA_OVERWRITE | USN_REASON_BASIC_INFO_CHANGE | USN_REASON_DATA_TRUNCATION | USN_REASON_CLOSE.

The change journal accumulates a series of records between the first opening and last closing of a file. Each record has a new reason flag set, indicating that a new kind of change has occurred. The sequence of records gives a partial history of the file. The final record, created when the file is closed, adds the USN_REASON_CLOSE flag. This record represents a summary of changes to the file, but unlike the prior records, gives no indication of the order of the changes.
The next user to access and change the file generates a new USN record with a single reason flag.

Using the Change Journal Identifier

The NTFS file system associates an unsigned 64-bit identifier with each change journal. The journal is stamped with this identifier when it is created. The file system stamps the journal with a new identifier where the existing USN records either are or may be unusable.
For example, the NTFS file system re-stamps a change journal with a new identifier when a volume is moved from Windows 2000 to Windows XP and then back to Windows 2000. Such a move can happen in a dual-boot environment or when working with removable media.
To obtain the identifier of the current change journal on a specified volume, use the FSCTL_QUERY_USN_JOURNAL control code. To perform this and all other change journal operations, you must have system administrator privileges. That is, you must be a member of the Administrators group.
When an administrator deletes and recreates the change journal, for example when the current USN value approaches the maximum possible USN value, the USN values begin again from zero. When the NTFS file system stamps a journal with a new identifier rather than recreating the journal, it does not reset the USN to zero but continues from the current USN. In either case, all existing USNs are less than any future USNs.
When you need information on a specific set of records, use the FSCTL_QUERY_USN_JOURNAL control code to obtain the change journal identifier. Then use the FSCTL_READ_USN_JOURNAL control code to read the journal records of interest. The NTFS file system only returns records that are valid for the journal specified by the identifier.
Your application needs both the records' USNs and the identifier to read the journal. This requirement provides an integrity check for cases where your application should ignore the existing records in the file and where records were written in previous instances of the journal for the same volume.
To obtain the records in which you are interested, you must start at the oldest record (that is, with the lowest USN) and scan forward until you locate the first record of interest.

Creating, Modifying, and Deleting a Change Journal

Administrators can create, delete, and recreate change journals at will. An administrator should delete a journal when the current USN value approaches the maximum possible USN value, as indicated by the MaxUsn member of the USN_JOURNAL_DATA structure. An administrator might also delete and recreate a change journal to reclaim disk space. To perform this and all other non-programmatic change journal operations, you must have system administrator privileges. That is, you must be a member of the Administrators group.
To create or modify a change journal on a specified volume programmatically, use the FSCTL_CREATE_USN_JOURNAL control code.

When you create a new change journal or modify an existing one, the NTFS file system sets information for that change journal from information in the CREATE_USN_JOURNAL_DATA structure, which FSCTL_CREATE_USN_JOURNAL takes as input.
CREATE_USN_JOURNAL_DATA has the members MaximumSize and AllocationDelta.
MaximumSize is the target maximum size for the change journal in bytes. The change journal can grow larger than this value, but at NTFS file system checkpoints the NTFS file system examines the journal and trims it when its size exceeds the value of MaximumSize plus the value of AllocationDelta. (At NTFS file system checkpoints, the operating system writes records to the NTFS file system log file that allow the NTFS file system to determine what processing is required to recover from a failure.)
AllocationDelta is the number of bytes added to the end and removed from the beginning of the change journal each time memory is allocated or deallocated. In other words, allocation and deallocation take place in units of this size. An integer multiple of a cluster size is a reasonable value for this member.
If an administrator modifies an existing change journal to have a larger MaximumSize value, for example if a volume is being re-indexed too often, the change journal simply receives new entries until it exceeds the new maximum size.
To delete a change journal, use the FSCTL_DELETE_USN_JOURNAL control code. When you use this operation, it walks through all of the files on the volume and resets the USN for each file to zero. The operation then deletes the existing change journal. This operation persists across system restarts until it completes. Any attempt to read, create, or modify the change journal during this process fails with the error code ERROR_JOURNAL_DELETE_IN_PROGRESS.
You can also use the FSCTL_DELETE_USN_JOURNAL control code to determine if a deletion started by some other process is in progress. For example, your application, when it is started, can determine if a deletion is in progress. Because journal deletions persist across system restarts, services and applications started at system restart should check for an ongoing deletion.
Change journals are not necessarily created at startup. To create a change journal, an administrator may do so explicitly or start another service that requires a change journal.

Obtaining a Volume Handle for Change Journal Operations

To obtain a handle to a volume for use with change journal operations, call the CreateFile() function with the lpFileName parameter set to a string of the following form: \\.*X*:
Note that X is the letter that identifies the drive on which the NTFS volume appears. If the volume does not have a drive letter, use the syntax described in Naming a Volume section.

Change Journal Operations

The following list identifies the control codes that work with the NTFS file system change journal.

1. FSCTL_CREATE_USN_JOURNAL
2. FSCTL_DELETE_USN_JOURNAL
3. FSCTL_ENUM_USN_DATA
4. FSCTL_MARK_HANDLE
5. FSCTL_QUERY_USN_JOURNAL
6. FSCTL_READ_USN_JOURNAL

The following list identifies the structures information that relates to the NTFS file system change journal.

1. CREATE_USN_JOURNAL_DATA
2. DELETE_USN_JOURNAL_DATA
3. MARK_HANDLE_INFO
4. MFT_ENUM_DATA
5. READ_USN_JOURNAL_DATA
6. USN_JOURNAL_DATA
7. USN_RECORD

Walking a Buffer of Change Journal Records

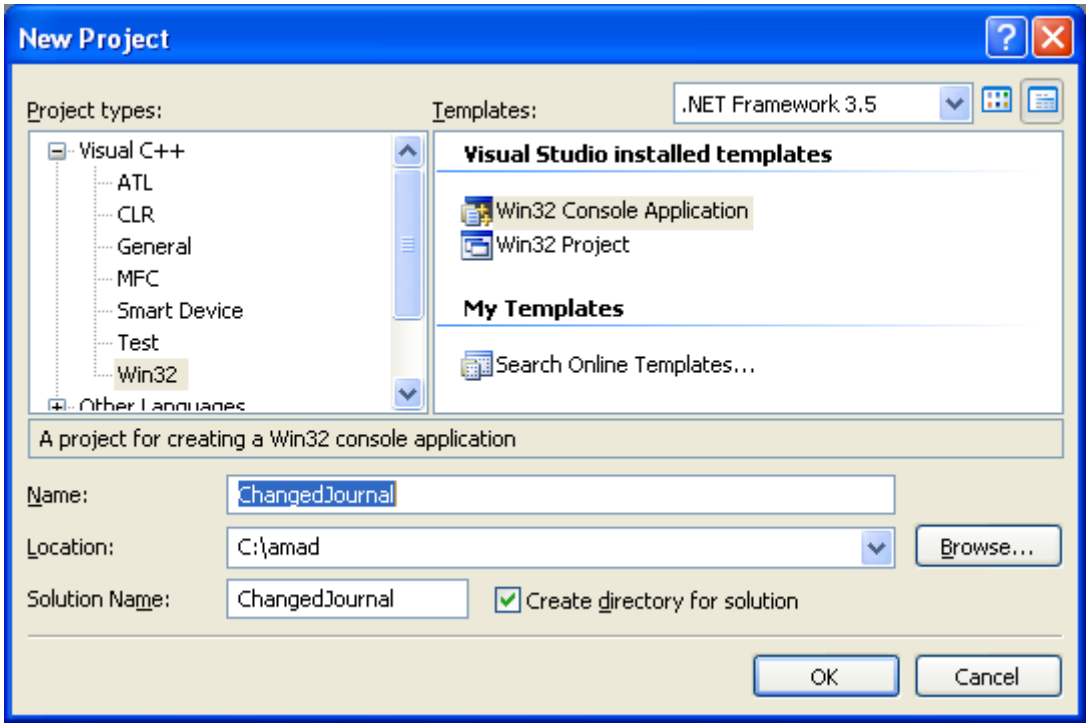
The control codes that return change journal records, FSCTL_READ_USN_JOURNAL and FSCTL_ENUM_USN_DATA, return similar data in the output buffer. Both return a USN followed by 0 (zero) or more change journal records, each in a USN_RECORD structure. The following list identifies ways to get change journal records:

1. Use FSCTL_ENUM_USN_DATA to get a listing (enumeration) of all change journal records between two USNs.
2. Use FSCTL_READ_USN_JOURNAL to be more selective, such as selecting specific reasons for changes or returning when a file is closed.

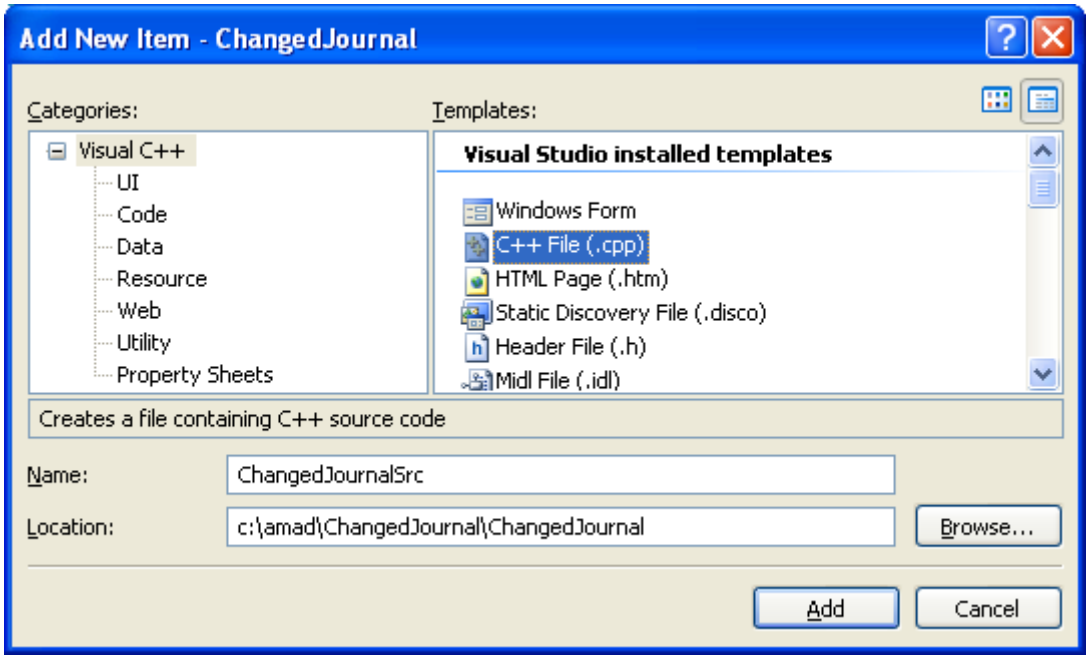
Both of these operations return only the subset of change journal records that meet the specified criteria.
The USN returned as the first item in the output buffer is the USN of the next record number to be retrieved. Use this value to continue reading records from the end boundary forward.
The FileName member of USN_RECORD contains the name of the file to which the record in question applies. The file name varies in length, so USN_RECORD is a variable length structure. Its first member, RecordLength, is the length of the structure (including the file name), in bytes.
When you work with the FileName member of USN_RECORD, do not assume that the file name contains a trailing '\0' delimiter. To determine the length of the file name, use the FileNameLength member.

Walking a Buffer of Change Journal Records Program Example

The following example calls FSCTL_READ_USN_JOURNAL and walks the buffer of change journal records that the operation returns. To compile an application that uses this function, you may need to define the _WIN32_WINNT macro as 0x0500 (Refer to Using the Windows Headers).
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

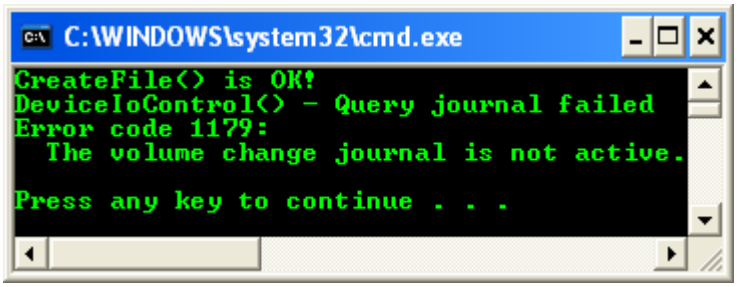
```
#include <Windows.h>
#include <WinIoCtl.h>
#include <stdio.h>
```

```
#define BUF_LEN 4096

// Format the Win32 system error code to string
void ErrorMessage(DWORD dwCode);

int wmain(int argc, WCHAR **argv)
{
   HANDLE hVol;
   CHAR Buffer[BUF_LEN];
   USN_JOURNAL_DATA JournalData;
   READ_USN_JOURNAL_DATA ReadData = {0, 0xFFFFFFFF, FALSE, 0, 0};
   PUSN_RECORD UsnRecord;
   DWORD dwBytes;
   DWORD dwRetBytes;
   int i;

   hVol = CreateFile( L"\\\\.\\C:",
               GENERIC_READ | GENERIC_WRITE,
               FILE_SHARE_READ | FILE_SHARE_WRITE,
               NULL,
               OPEN_EXISTING,
               0,
               NULL);

   if(hVol == INVALID_HANDLE_VALUE)
   {
      wprintf(L"CreateFile() failed\n");
        ErrorMessage(GetLastError());
      return 1;
   }

   wprintf(L"CreateFile() is OK!\n");

   if(!DeviceIoControl(hVol,
          FSCTL_QUERY_USN_JOURNAL,
          NULL,
          0,
          &JournalData,
          sizeof(JournalData),
          &dwBytes,
          NULL))
   {
      wprintf(L"DeviceIoControl() - Query journal failed\n");
        ErrorMessage(GetLastError());
      return 1;
   }

   ReadData.UsnJournalID = JournalData.UsnJournalID;

   wprintf(L"Journal ID: %I64x\n", JournalData.UsnJournalID );
   wprintf(L"FirstUsn: %I64x\n\n", JournalData.FirstUsn );

   for(i=0; i<=10; i++)
   {
      memset(Buffer, 0, BUF_LEN);
```

```
      if(!DeviceIoControl( hVol,
            FSCTL_READ_USN_JOURNAL,
            &ReadData,
            sizeof(ReadData),
            &Buffer,
            BUF_LEN,
            &dwBytes,
            NULL))
      {
         wprintf(L"DeviceIoControl()- Read journal failed\n");
             ErrorMessage(GetLastError());
         return 1;
      }

        wprintf(L"DeviceIoControl() is OK!\n");

      dwRetBytes = dwBytes - sizeof(USN);

      // Find the first record
      UsnRecord = (PUSN_RECORD)(((PUCHAR)Buffer) + sizeof(USN));

      printf( "*************************************\n");

      // This loop could go on for a long time, given the current buffer size.
      while(dwRetBytes > 0)
      {
         wprintf(L"USN: %I64x\n", UsnRecord->Usn );
         wprintf(L"File name: %.*S\n", UsnRecord->FileNameLength/2,UsnRecord-
>FileName );
         wprintf(L"Reason: %x\n", UsnRecord->Reason );
         wprintf(L"\n" );

         dwRetBytes -= UsnRecord->RecordLength;

         // Find the next record
         UsnRecord = (PUSN_RECORD)(((PCHAR)UsnRecord) + UsnRecord-
>RecordLength);
      }
      // Update starting USN for next call
      ReadData.StartUsn = *(USN *)&Buffer;
   }

   if(CloseHandle(hVol) != 0)
         wprintf(L"CloseHandle() is OK!\n");
   else
   {
         wprintf(L"CloseHandle() failed\n");
         ErrorMessage(GetLastError());
   }
   return 0;
}

void ErrorMessage(DWORD dwCode)
{
    // get the error code...
    DWORD dwErrCode = dwCode;
```

```
      DWORD dwNumChar;

    LPWSTR szErrString = NULL;  // will be allocated and filled by FormatMessage

    dwNumChar = FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM, // use windows internal message
table
                  0,       // 0 since source is internal message table
                  dwErrCode, // this is the error code number
                  0,       // auto-determine language to use
                  (LPWSTR)&szErrString, // the messsage
                  0,                 // min size for buffer
                  0 );               // since getting message from system tables
      if(dwNumChar == 0)
            wprintf(L"FormatMessage() failed, error %u\n", GetLastError());
      //else
      //    wprintf(L"FormatMessage() should be fine!\n");

     wprintf(L"Error code %u:\n  %s\n", dwErrCode, szErrString) ;

      // This buffer used by FormatMessage()
    if(LocalFree(szErrString) != NULL)
            wprintf(L"Failed to free up the buffer, error %u\n",
GetLastError());
      //else
      //    wprintf(L"Buffer has been freed\n");

  }
```

Build and run the project. The following screenshot is an output sample and there is no active journal in the system.



The size in bytes of any record specified by a USN_RECORD structure is at most ((MaxComponentLength - 1) * Width) + Size where MaxComponentLength is the maximum length in characters of the record file name. The width is the size of a wide character, and the Size is the size of the structure.
To obtain the maximum length, call the GetVolumeInformation() function and examine the value pointed to by the lpMaximumComponentLength parameter. Subtract one from MaxComponentLength to account for the fact that the definition of USN_RECORD includes one character of the file name. In the C programming language, the largest possible record size is the following:

```
MaxComponentLength*sizeof(WCHAR) + sizeof(USN_RECORD) -
sizeof(WCHAR)
```

Mounted Folders (drives)

The NTFS file system supports mounted folders. A mounted folder is an association between a volume and a directory on another volume. When a mounted folder is created, users and applications can access the target volume either by using the path to the mounted folder or by using the volume's drive letter. For example, a user can create a mounted folder to associate drive D: with the C:\Mnt\DDrive folder on drive C. After creating the mounted folder, the user can use the "C:\Mnt\DDrive" path to access drive D: as if it were a folder on drive C:.
Using mounted folders, you can unify disparate file systems such as the NTFS file system, a 16-bit FAT file system, and an ISO-9660 file system on a CD-ROM drive into one logical file system on a single NTFS volume. Neither users nor applications need information about the target volume on which a specific file is located. All the information they need to locate a specified file is a complete path using a mounted folder on the NTFS volume. Volumes can be rearranged, substituted, or subdivided into many volumes without users or applications needing to change settings.

How to create a mounted drive

To mount a volume:

1. Click **Start**, click **Run**, and then type compmgmt.msc in the **Open** box or open the Computer Management snap-in from Administrative Tools (Windows XP Pro SP2).
2. In the left pane, click **Disk Management**.
3. Right-click the partition or volume that you want to mount, and then click **Change Drive Letter and Paths**.
4. Click **Add**.
5. Click **Mount in the following empty NTFS folder** (if it is not already selected), and then use one of the following steps:

    - Type the path to an empty folder on an NTFS volume, and then click **OK**.
    - Click **Browse**, locate the empty NTFS folder, click **OK**, and then click **OK**.
    - If you have not yet created an empty folder, click **Browse**, click **New Folder** to create an empty folder on an NTFS volume, type a name for the new folder, click **OK**, and then click **OK**.

6. Quit the Disk Management snap-in.

How to remove a mounted drive

To remove a mounted volume:

1. Click **Start**, click **Run**, and then type compmgmt.msc in the **Open** box or open the Computer Management snap-in from other short-cut.
2. In the left pane, click **Disk Management**.

3. Right-click the partition or volume that you want to unmount, and then click **Change Drive Letter and Paths**.
4. Click the mounted drive path that you want to remove, and then click **Remove**.
5. Click **Yes** when you are prompted to remove the drive path.
6. Quit the Disk Management snap-in.

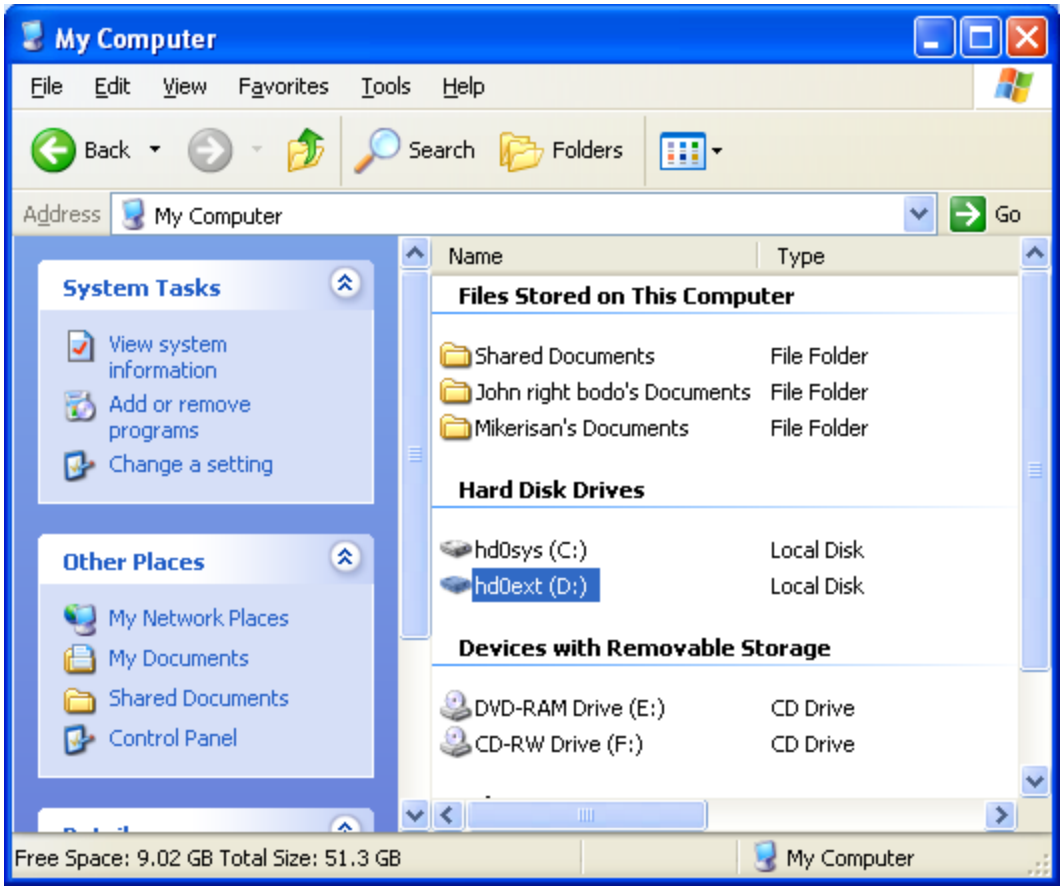
When you attempt to mount a volume on a folder on an NTFS volume, you may receive the following error message:

> *The folder you specified is not empty. A volume can be mounted only at an empty folder.*

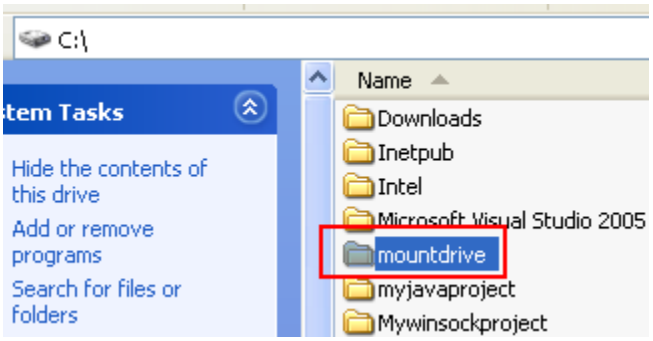
This message is displayed when the folder in which you want to mount the volume is not empty. To resolve this issue, create a new empty folder in which to mount the volume, or delete the contents of the folder, and then mount the volume.
When you attempt to mount a volume on a folder on an NTFS volume, you may receive the following error message:

> *The path provided is on a file system that does not support drive paths.*

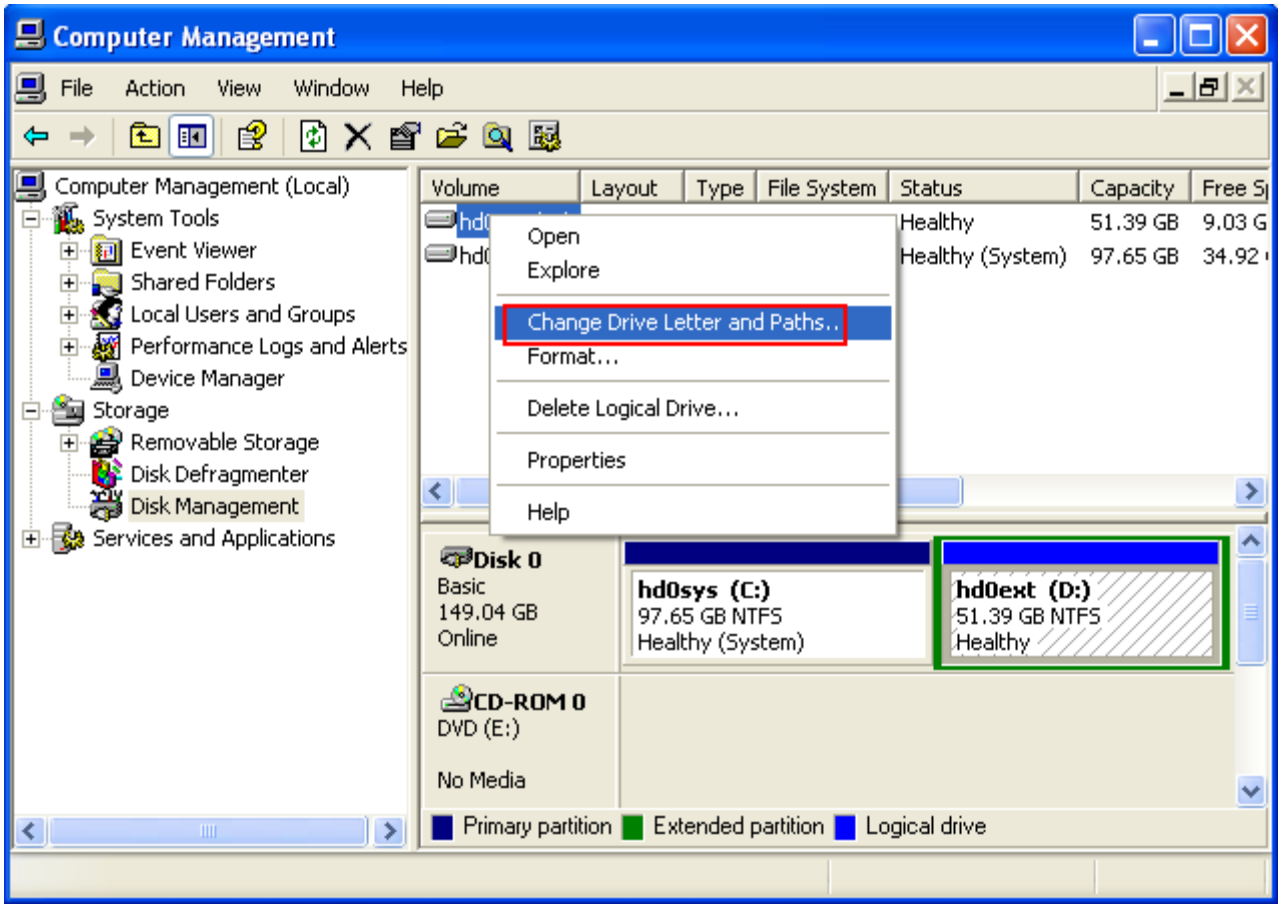
This message is displayed if the volume is not formatted with the NTFS file system. To resolve this issue, make sure that the volume in which you want to host the mounted drive is an NTFS volume.
The following steps show how to mount a drive to an empty folder. We will mount D: drive to an empty folder, mountdrive.



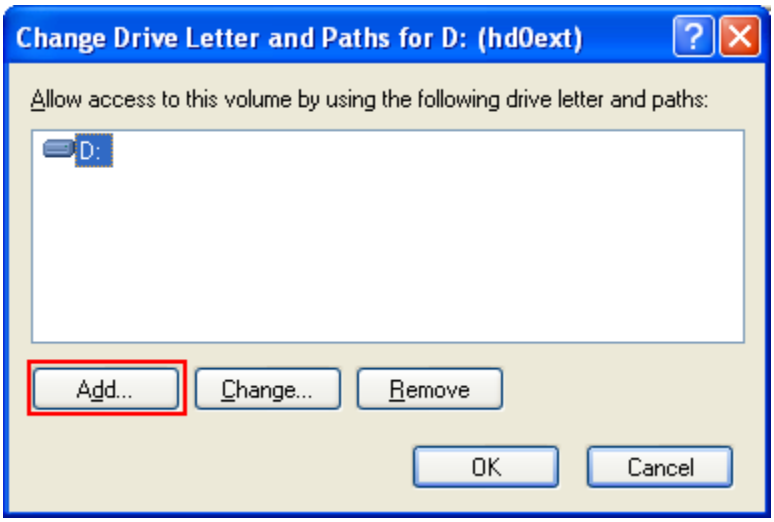
Firstly we create an empty folder named mountdrive on drive C:.



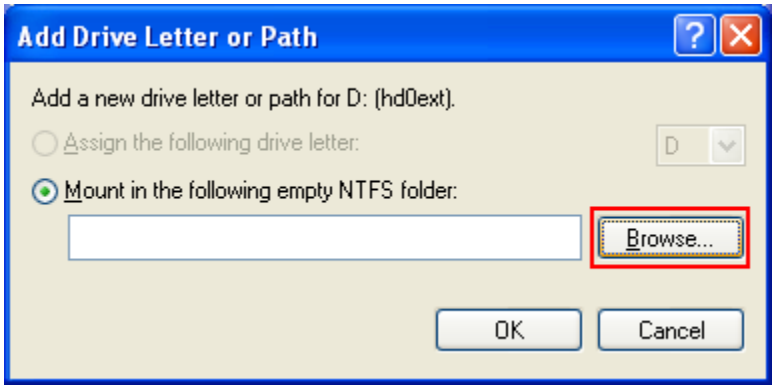
Next, through Computer Management snap-in, we invoke the Change Drive Letter and Paths page while selecting D: drive.



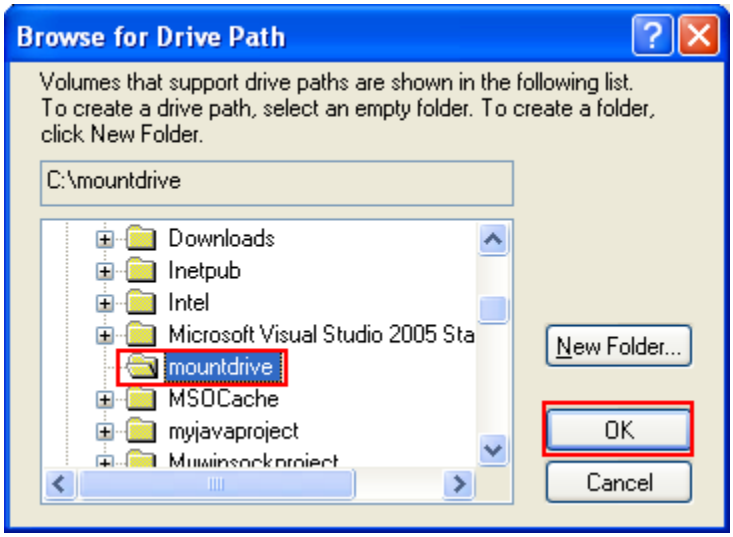
Next, click the Add button.



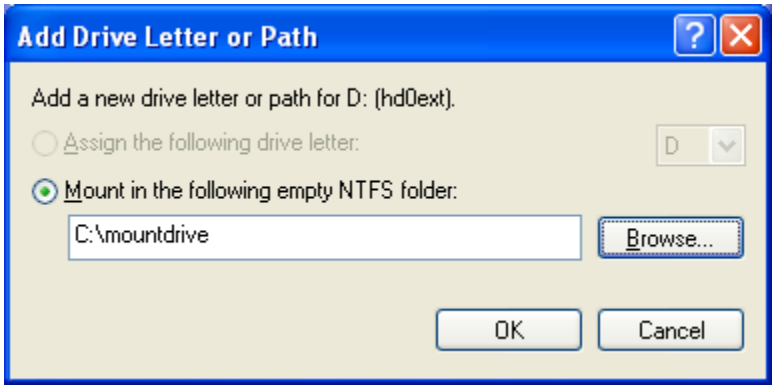
Select the second radio button, Mount in the following empty NTFS folder. Then click the Browse button to browse the folder that we will mount this drive. In this case it is the mountdrive created previously.



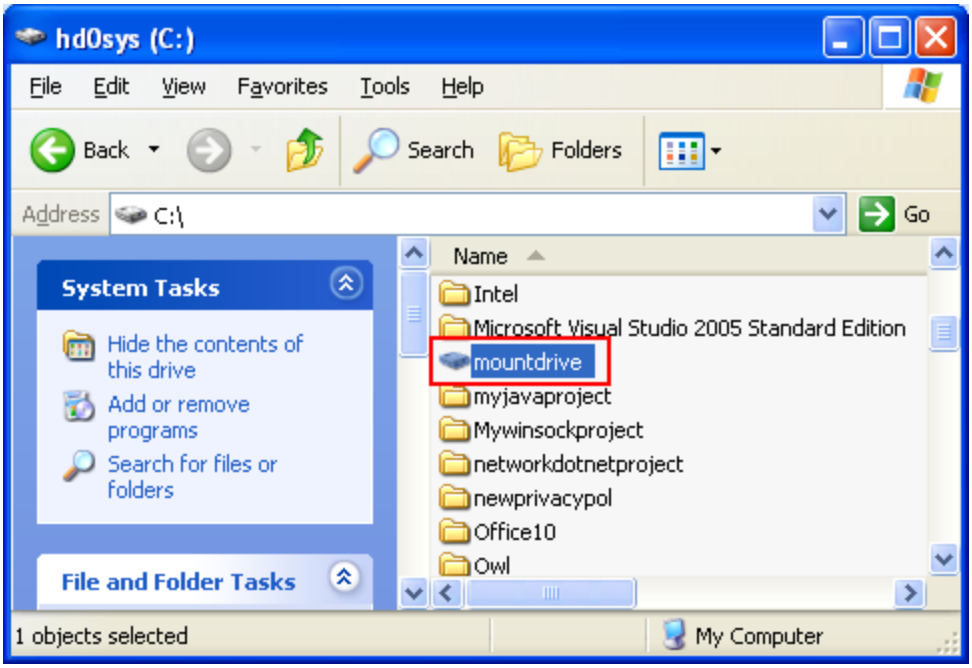
Click the mountdrive and click OK button.



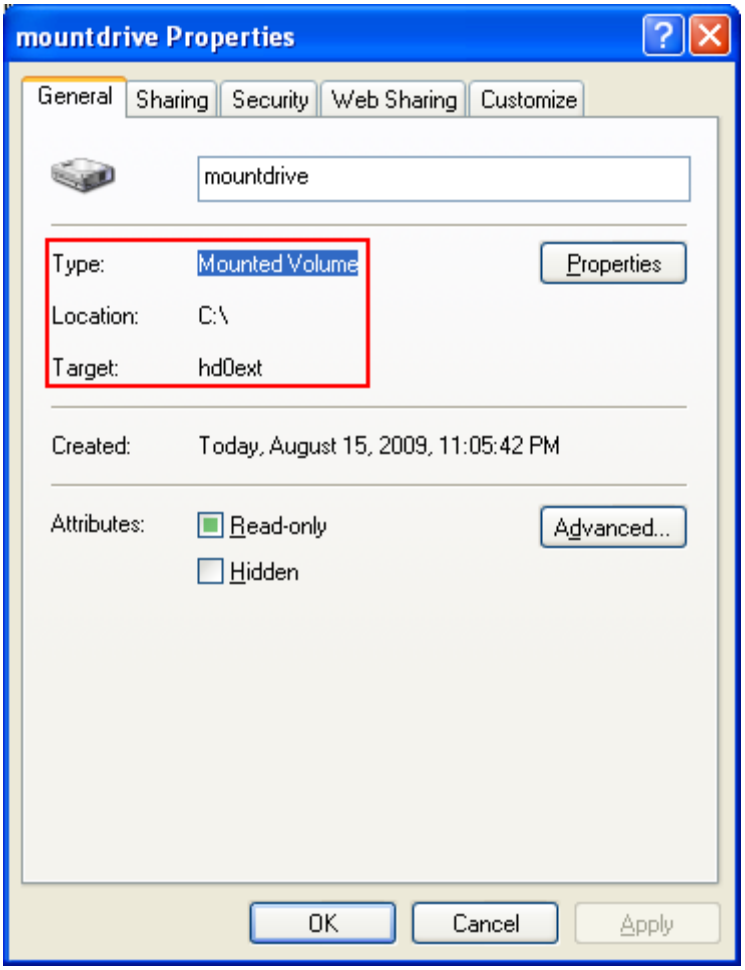
Click OK button to complete the tasks.



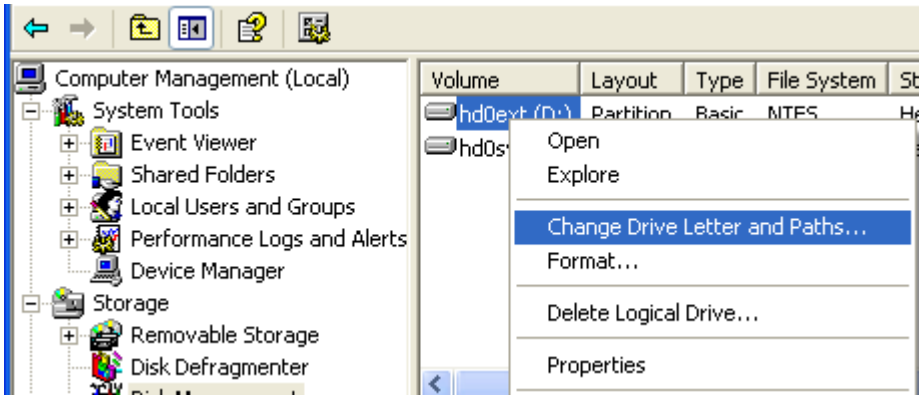
The following Figure shows the mounted drive through Windows Explorer.



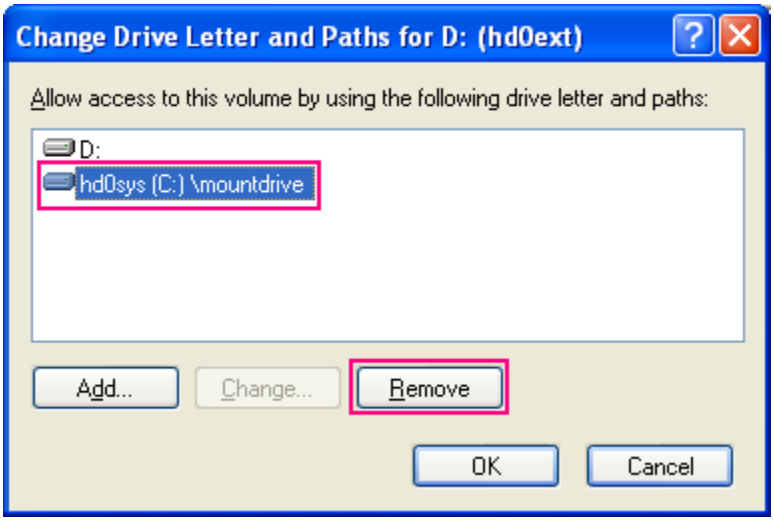
The mounted drive properties page is shown below.



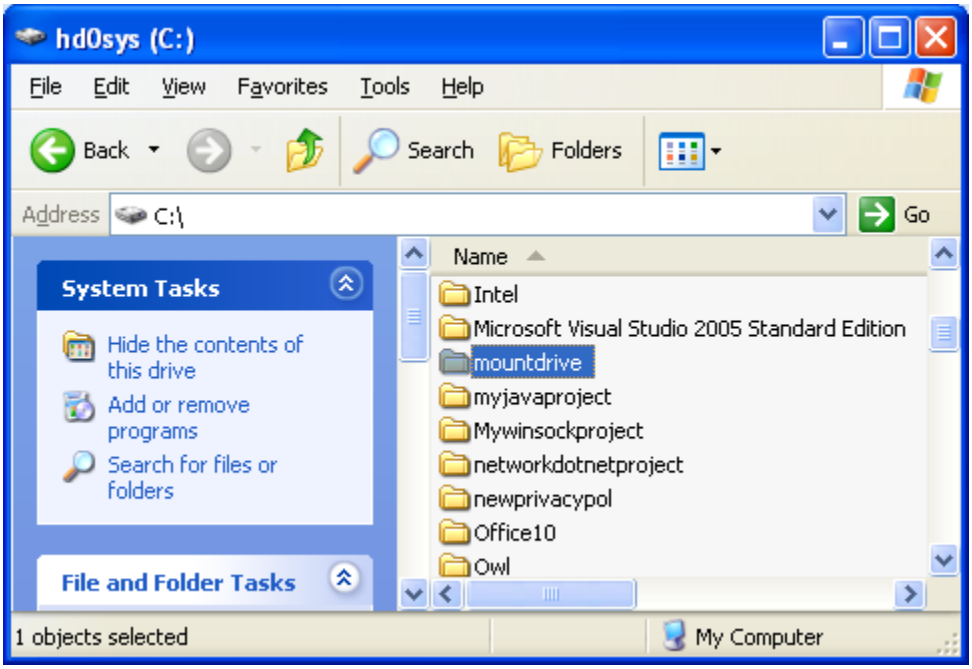
To remove the mounted drive, invoke the same menu.



Select the mounted drive and click Remove button.



The folder is back to a normal folder.



Creating Mounted Folders Programmatically

Creating a mounted folder is a two-step process. First, we call GetVolumeNameForVolumeMountPoint() with the mount point (drive letter, volume GUID path, or mounted folder) of the volume to be assigned to the mounted folder. Then use the SetVolumeMountPoint() function to associate the returned volume GUID path with the desired directory on another volume.
Your application can designate any empty directory on a volume other than the root as a mounted folder. When you call the SetVolumeMountPoint() function, that directory becomes the mounted folder. You can assign the same volume to multiple mounted folders.

After the mounted folder has been established, it is maintained through computer restarts automatically.
If a volume fails, any volumes that have been assigned to mounted folders on that volume can no longer be accessed through those mounted folders. For example, suppose you have two volumes, C: and D:, and that D: is associated with the mounted folder C:\MountD\. If volume C: fails, volume D: can no longer be accessed through the path C:\MountD\.
Only NTFS file system volumes can have mounted folders, but the target volumes for the mounted folders can be non-NTFS volumes.
Mounted folders are implemented by using reparse points and are subject to their restrictions. It is not necessary to manipulate reparse points to use mounted folders; functions such as SetVolumeMountPoint() handle all the reparse point details for you.
Because mounted folders are directories, you can rename, remove, move, and otherwise manipulate them, as you would other directories. (Note: The TechNet documentation uses the term *mounted drives* to refer to *mounted folders*.)

Enumerating Mounted Folders

The following functions are used to enumerate the mounted folders on a specified NTFS volume:

1. FindFirstVolumeMountPoint()
2. FindNextVolumeMountPoint()
3. FindVolumeMountPointClose()

These functions operate in a manner very similar to the FindFirstFile(), FindNextFile(), and FindClose() functions.
To enumerate mounted folders on a volume, first find out if the volume supports mounted folders. To do so, use the volume name returned by the FindFirstVolume() and FindNextVolume() functions to call the GetVolumeInformation() function. The names returned include a trailing backslash (\) to be compatible with the GetDriveType() function and related functions. When you call the GetVolumeInformation() function, if "NTFS" is returned in the *lpFileSystemNameBuffer* parameter, the volume is an NTFS volume. The NTFS file system supports mounted folders.
If the volume is an NTFS volume, begin a search for the mounted folders by calling FindFirstVolumeMountPoint(). If the search is successful, process the results according to your application's requirements. Then use FindNextVolumeMountPoint() in a loop to locate and process the mounted folders one at a time. When there are no more mounted folders to be enumerated, close the search handle by calling FindVolumeMountPointClose(). Note that the search will find only the mounted folders that are on the specified volume.
You should not assume any correlation between the order of the mounted folders that are returned by these functions and the order of the mounted folders that are returned by other functions or tools.

Determining Whether a Directory Is a Mounted Folder

It is useful to determine whether a directory is a mounted folder when, for example, you are using a backup or search application that is limited to one volume. Such an application can reach information on multiple volumes if you use functions such as SetVolumeMountPoint() to create mounted folders for the other volumes on the volume that the application is limited to.

To determine if a specified directory is a mounted folder, first call the GetFileAttributes() function and inspect the FILE_ATTRIBUTE_REPARSE_POINT flag in the return value to see if the directory has an associated reparse point. If it does, use the FindFirstFile() and FindNextFile() functions to obtain the reparse tag in the **dwReserved0** member of the WIN32_FIND_DATA structure. To determine if the reparse point is a mounted folder (and not some other form of reparse point), test whether the tag value equals the value IO_REPARSE_TAG_MOUNT_POINT.
To obtain the target volume of a mounted folder, use the GetVolumeNameForVolumeMountPoint() function. In a similar manner, you can determine if a reparse point is a symbolic link by testing whether the tag value is IO_REPARSE_TAG_SYMLINK.

Assigning a Drive Letter to a Volume

You can assign a drive letter (for example, x:\) to a local volume using SetVolumeMountPoint(), provided there is no volume already assigned to that drive letter. If the local volume already has a drive letter then SetVolumeMountPoint**()** will fail. To handle this, first delete the drive letter using DeleteVolumeMountPoint(). The system supports at most one drive letter per volume. Therefore, you cannot have C:\ and F:\ represent the same volume.

Caution

Deleting an existing drive letter and assigning a new one may break existing paths, such as those in desktop shortcuts. It may also break the path to the program making the drive letter changes. With Windows virtual memory management, this may break the application, leaving the system in an unstable and possibly unusable state. It is the program designer's responsibility to avoid such potential catastrophes.

Mounted Folder Functions

The mounted folder functions can be divided into three groups:

1. General-purpose functions
2. Functions used to scan for volumes, and
3. Functions used to scan a volume for mounted folders.

General-Purpose Mounted Folder Functions

The following Table lists the general-purpose mounted folder functions.

Function	Description
DeleteVolumeMountPoint()	Deletes a drive letter or mounted folder.
GetVolumeNameForVolumeMountPoint()	Retrieves the volume GUID path for the volume that is associated with the specified volume mount point (drive letter, volume GUID path, or mounted folder).
GetVolumePathName()	Retrieves the mounted folder that is associated with the specified volume.

SetVolumeMountPoint()	Associates a volume with a drive letter or a directory on another volume.

Volume-Scanning Functions

The following Table lists the volume-scanning functions.

Function	Description
FindFirstVolume()	Returns the name of a volume on a computer. FindFirstVolume() is used to begin enumerating the volumes of a computer.
FindNextVolume()	Continues a volume search started by a call to FindFirstVolume().
FindVolumeClose()	Closes a search for volumes.

Mounted Folder Scanning Functions

The following Table lists the mounted folder scanning functions.

Function	Description
FindFirstVolumeMountPoint()	Retrieves the name of a mounted folder on the specified volume. FindFirstVolumeMountPoint() is used to begin scanning the mounted folders on a volume.
FindNextVolumeMountPoint()	Continues a mounted folder search started by a call to FindFirstVolumeMountPoint().
FindVolumeMountPointClose()	Closes a search for mounted folders.

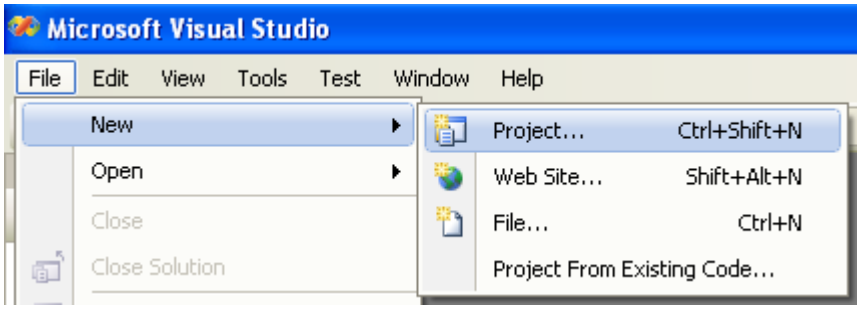
Mounted Folder Program Examples

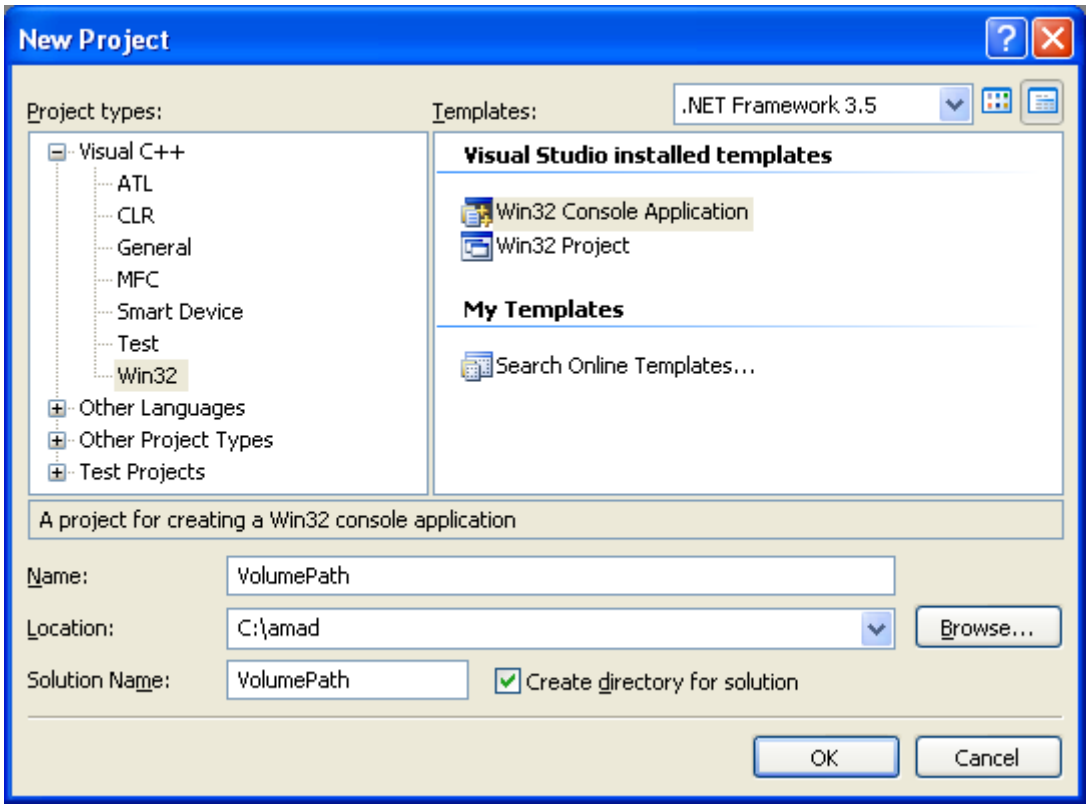
The following examples illustrate the mounted folder functions which include:

1. Displaying Volume Paths
2. Editing Drive Letter Assignments
3. Creating a Mounted Folder
4. Enumerating Volume GUID Paths
5. Deleting a Mounted Folder

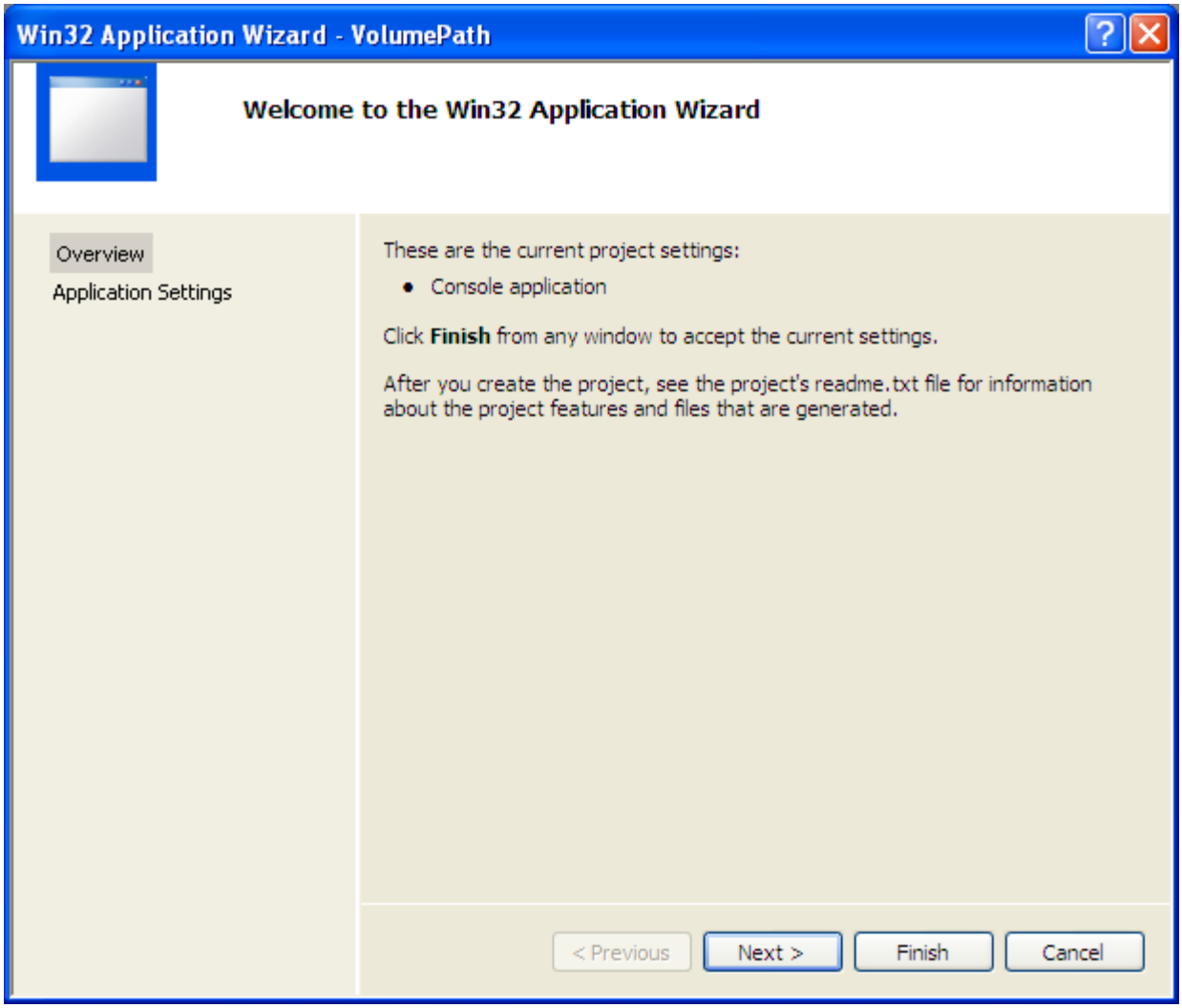
Displaying Volume Paths Program Example

The following example shows how to display all paths for each volume and device. For each volume in the system, the example locates the volume, obtains the device name, obtains all paths for that volume, and displays the paths.
Create a new Win32 console application project and give a suitable project name.

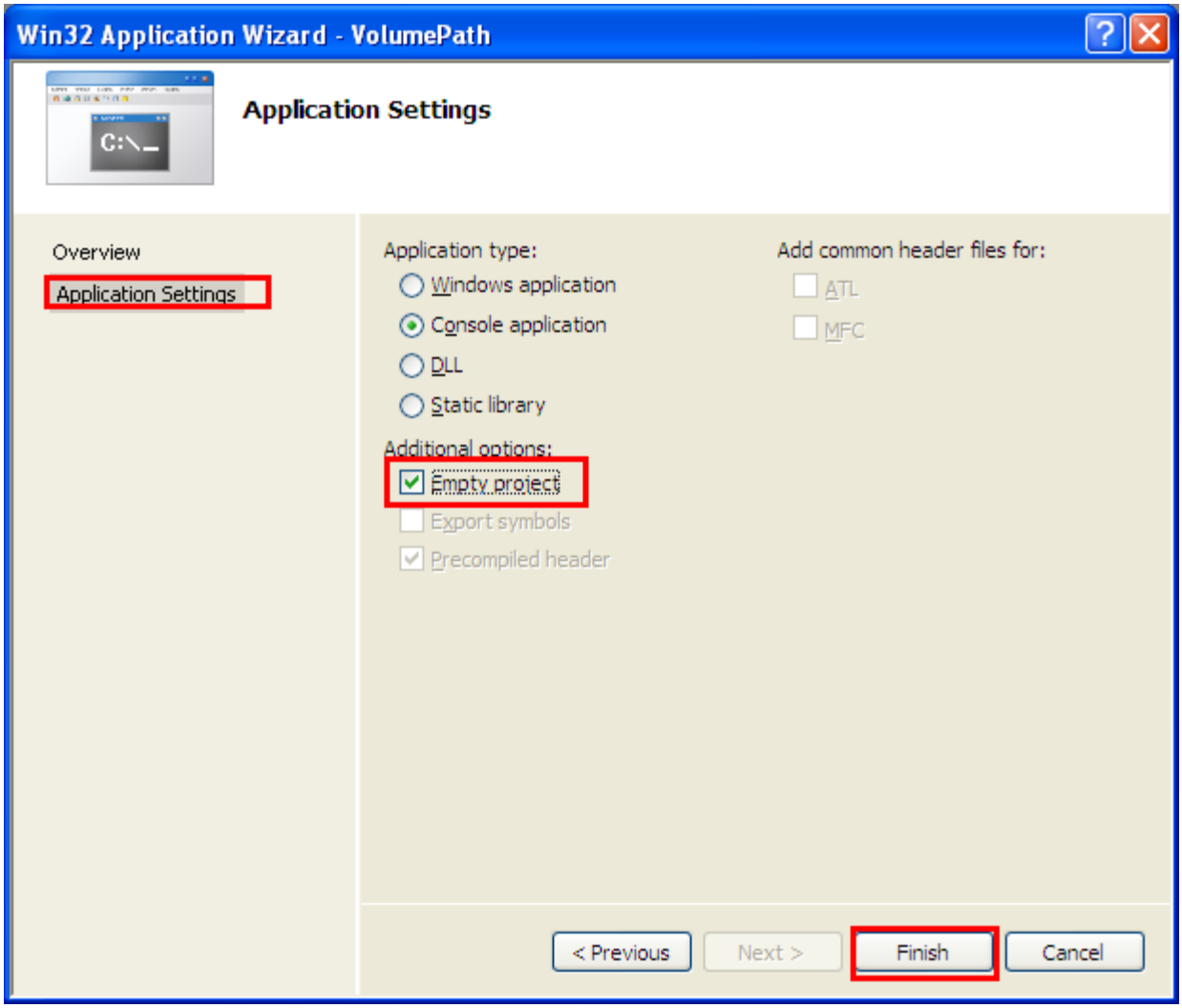




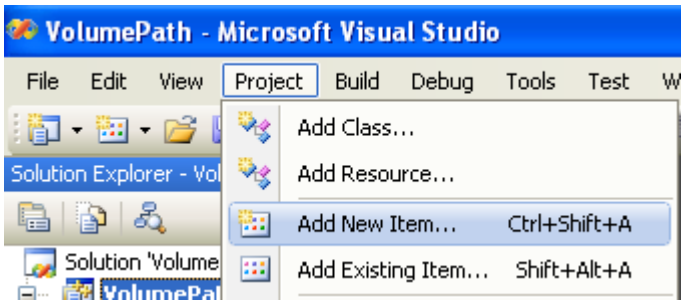
Select the Next button to refine the project template properties.



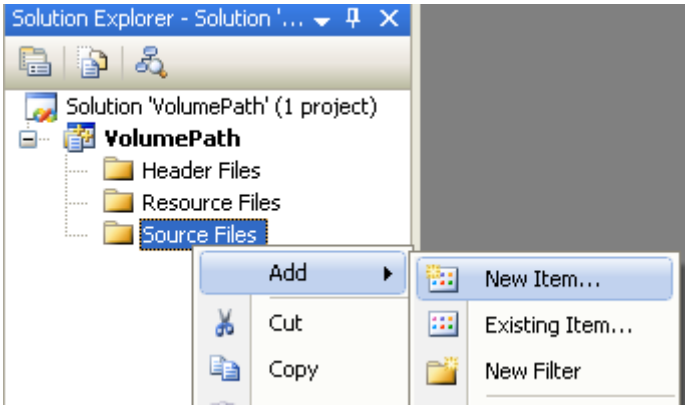
Select the Empty project tick box and leave others as if. Click the Finish button.



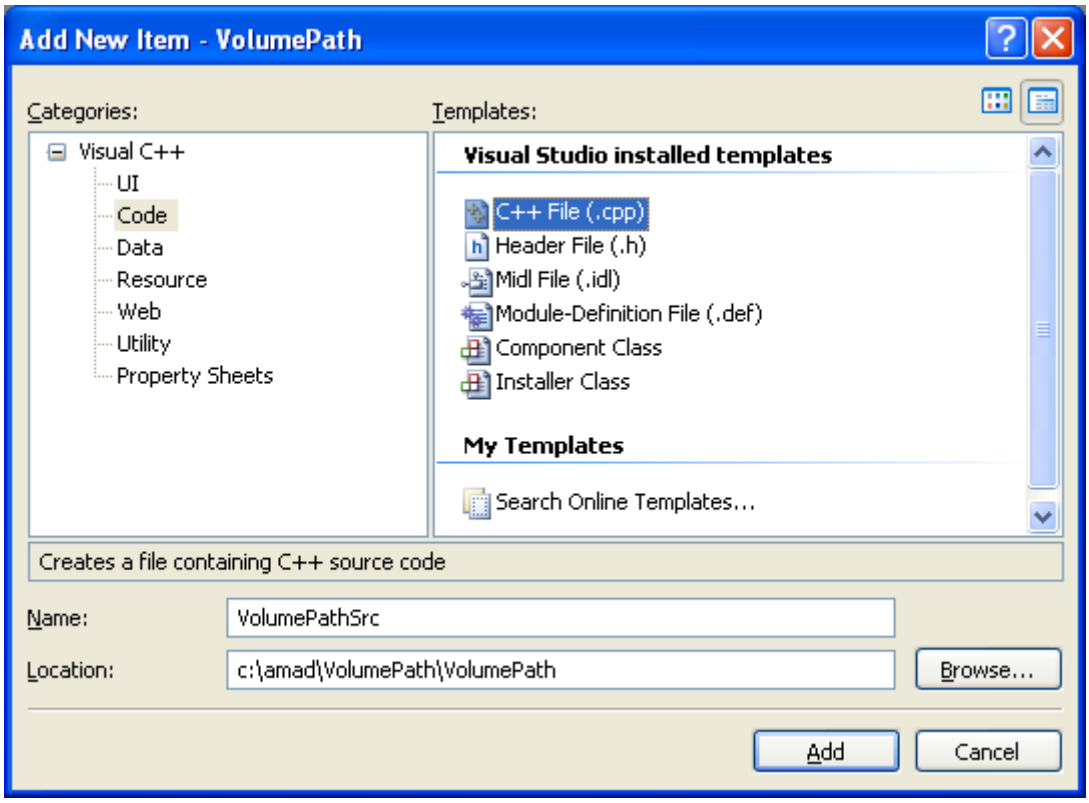
Add the source file and give a suitable name. Select Project > Add New Item menu.



Or, select the Source Files folder in Solution Explorer > Right click mouse > select Add menu > select New Item sub menu.



Select Code for Categories: and C++ File (.cpp) for Templates:. Give a suitable source file name.



Add the following source code. Build and run the project.

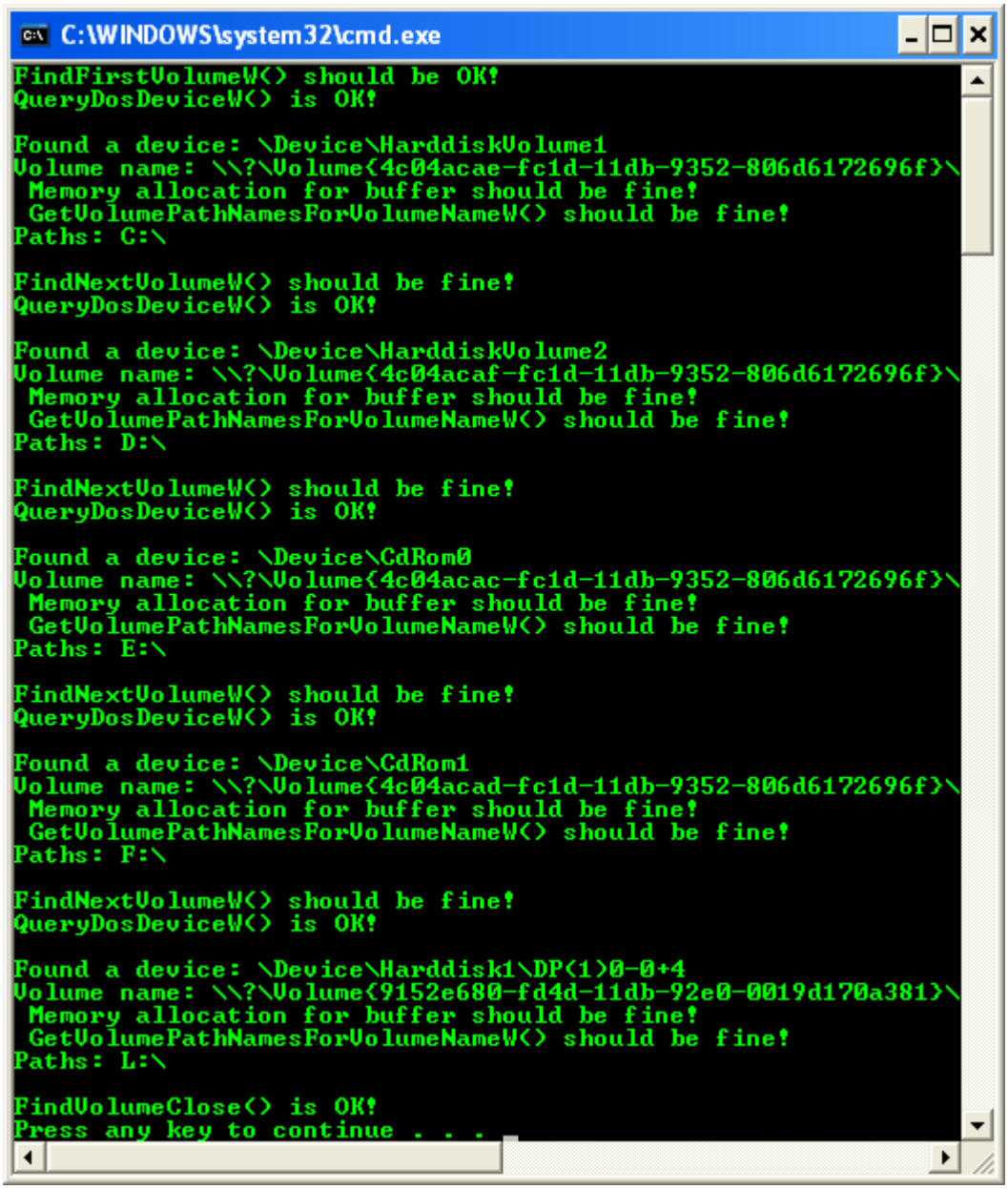
```
// Compile As C++ Code (/TP) and using Unicode character set
#include <windows.h>
#include <stdio.h>

PWCHAR DisplayVolumePaths(PWCHAR VolumeName)
{
    DWORD  CharCount = MAX_PATH + 1;
    PWCHAR Names     = NULL;
    PWCHAR NameIdx   = NULL;
    BOOL   Success   = FALSE;
```

```
    for (;;)
    {
        //  Allocate a buffer to hold the paths.
        Names = (PWCHAR) new BYTE[CharCount * sizeof(WCHAR)];

        if (!Names)
        {
            //  If memory can't be allocated, return.
                  wprintf(L"\n Failed to allocate memory!\n");
            return L"Failed!";
        }

            wprintf(L"\n Memory allocation for buffer should be fine!");
        //  Obtain all of the paths for this volume.
        Success = GetVolumePathNamesForVolumeNameW(VolumeName, Names, CharCount,
&CharCount);

        if (Success)
        {
                  wprintf(L"\n GetVolumePathNamesForVolumeNameW() should be
fine!\n");
                  // Break the loop
            break;
        }

        if (GetLastError() != ERROR_MORE_DATA)
        {
                  wprintf(L"\n GetVolumePathNamesForVolumeNameW() failed with
error code %d\n", GetLastError());
            break;
        }

        //  Try again with the new suggested size
        delete [] Names;
        Names = NULL;
    }

    if (Success)
    {
        //  Display the various paths
        for (NameIdx = Names; NameIdx[0] != L'\0'; NameIdx += wcslen(NameIdx) +
1 )
        {
            return NameIdx;
        }
        wprintf(L"\n");
    }

      // Return the allocated buffer to the system
    if (Names != NULL)
    {
        delete [] Names;
        Names = NULL;
    }
```

```
    return L"Failed";
}

void wmain(void)
{
    DWORD  CharCount            = 0;
    WCHAR  DeviceName[MAX_PATH] = L"";
    DWORD  Error                = ERROR_SUCCESS;
    HANDLE FindHandle           = INVALID_HANDLE_VALUE;
    BOOL   Found                = FALSE;
    size_t Index                = 0;
    BOOL   Success              = FALSE;
    WCHAR  VolumeName[MAX_PATH] = L"";
      PWCHAR ret = L"";

    //  Enumerate all volumes in the system.
    FindHandle = FindFirstVolumeW(VolumeName, ARRAYSIZE(VolumeName));

    if (FindHandle == INVALID_HANDLE_VALUE)
    {
        Error = GetLastError();
        wprintf(L"FindFirstVolumeW() failed with error code %d\n", Error);
        return;
    }
      else
            wprintf(L"FindFirstVolumeW() should be OK!\n");

    for (;;)
    {
        //  Skip the \\?\ prefix and remove the trailing backslash.
        Index = wcslen(VolumeName) - 1;

        if (VolumeName[0]     != L'\\' ||
            VolumeName[1]     != L'\\' ||
            VolumeName[2]     != L'?'  ||
            VolumeName[3]     != L'\\' ||
            VolumeName[Index] != L'\\')
        {
            Error = ERROR_BAD_PATHNAME;
            wprintf(L"FindFirstVolumeW/FindNextVolumeW() returned a bad path:
%s\n", VolumeName);
            break;
        }

            // Retrieves information about MS-DOS device names.
        //  QueryDosDeviceW() doesn't allow a trailing backslash, so temporarily
remove it.
        VolumeName[Index] = L'\0';

        CharCount = QueryDosDeviceW(&VolumeName[4], DeviceName,
ARRAYSIZE(DeviceName));

        VolumeName[Index] = L'\\';

            // Returned value = 0 means failed
        if ( CharCount == 0 )
```

```
        {
            wprintf(L"QueryDosDeviceW() failed with error code %d\n",
GetLastError());
            break;
        }
            else
                  wprintf(L"QueryDosDeviceW() is OK!\n");

        wprintf(L"\nFound a device: %s", DeviceName);
        wprintf(L"\nVolume name: %s", VolumeName);
            ret = DisplayVolumePaths(VolumeName);
        wprintf(L"Paths: %s\n", ret);

        //  Move on to the next volume.
        Success = FindNextVolumeW(FindHandle, VolumeName,
ARRAYSIZE(VolumeName));

        if (!Success)
        {
            if (GetLastError() != ERROR_NO_MORE_FILES)
            {
                wprintf(L"\nFindNextVolumeW() failed with error code %d\n",
GetLastError());
                break;
            }

            //  Finished iterating through all the volumes.
            Error = ERROR_SUCCESS;
            break;
        }
            else
                  wprintf(L"\nFindNextVolumeW() should be fine!\n");
    }

      // Closes the volume search handle
    if( FindVolumeClose(FindHandle) == 0)
            wprintf(L"\nFindVolumeClose() failed with error code %s\n",
GetLastError());
      else
            wprintf(L"\nFindVolumeClose() is OK!\n");

    return;
}
```

The following is example output from running the application. For each volume, the output includes a volume device path, a volume GUID path, and a drive letter.



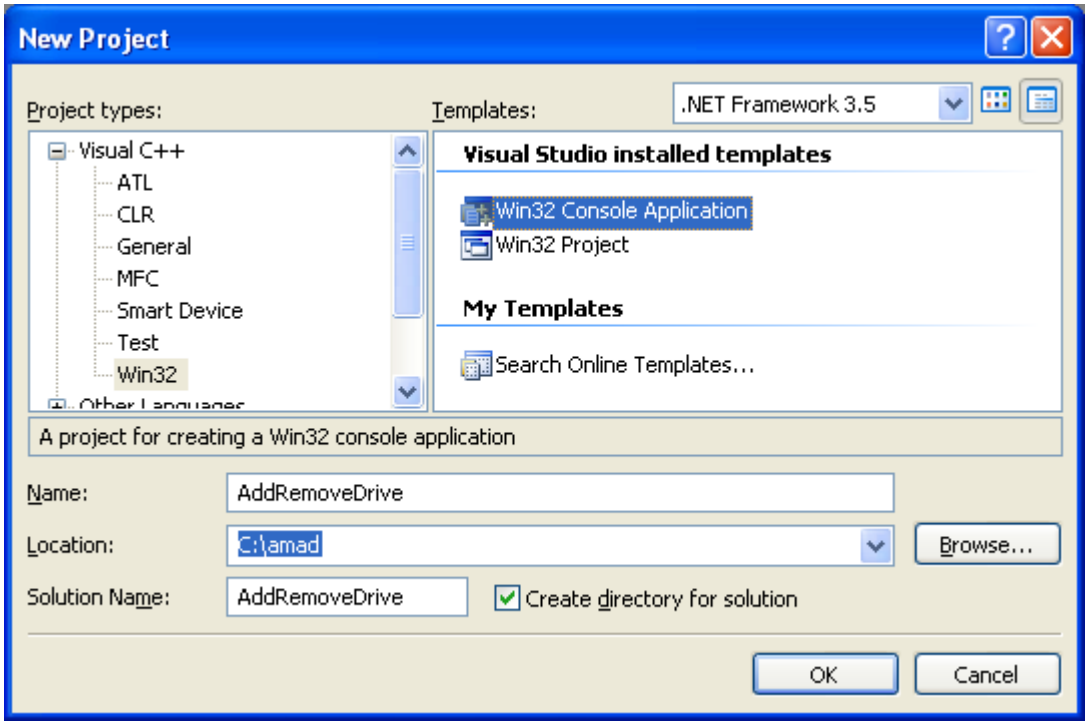
Editing Drive Letter Assignments Program Example

The code example in this topic shows you how to add or remove persistent drive letter assignments. These drive letter assignments persist through system shutdown. You are not encouraged to try this code on your personal PC!

The code example uses the following functions: DefineDosDevice(), DeleteVolumeMountPoint(), GetVolumeNameForVolumeMountPoint(), and SetVolumeMountPoint().

Create a new Win32 console application project and give a suitable project name.

Add the source file and give a suitable name.



Add the following source code.

```
/*
DLEDIT  -- Drive Letter Assignment Editor
Platforms: This program requires Windows 2000 or later.

Command-line syntax:
   DLEDIT <drive letter> <device name>      -- Adds a drive letter
   DLEDIT -r <drive letter>                 -- Removes a drive letter

Command-line examples:
   If E: refers to the CD-ROM drive, use the following commands to
   make F: point to the CD-ROM drive instead.

   DLEDIT -r E:\
   DLEDIT F:\ \Device\CdRom0

****************************************************************
WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING

   This program will change drive letter assignments, and the
   changes persist through reboots. Do not remove drive letters
   of your hard disks if you do not have this program on floppy
   disk or you might not be able to access your hard disks again!
****************************************************************
*/
// Windows Server 2003, Windows XP, change accordingly
// http://msdn.microsoft.com/en-us/library/aa383745(VS.85).aspx
#define _WIN32_WINNT 0x0501

#include <Windows.h>
```

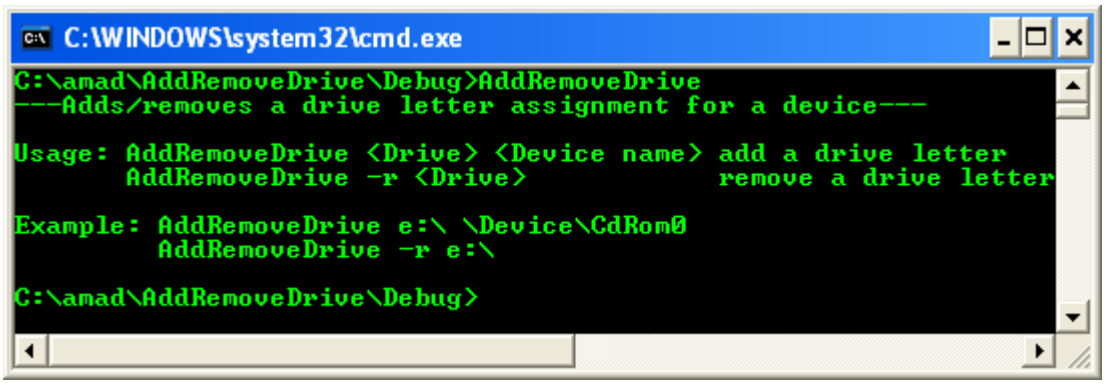
```
#include <stdio.h>

// For debug info
#if defined (DEBUG)
   static void DebugPrint (LPCTSTR pszMsg, DWORD dwErr);
   #define DEBUG_PRINT(pszMsg, dwErr) DebugPrint(pszMsg, dwErr)
#else
   #define DEBUG_PRINT(pszMsg, dwErr) NULL
#endif

// Disable the warning 4800 (forcing value to 'true' or 'false' (performance
warning)).
#pragma warning (disable : 4800)
// Other examples:
// Displaying the warning only once - #pragma warning ( once : 4800 )
// Apply the warning level (1-4) to the specified warning message(s) - #pragma
warning ( 3 : 4800 )
// Report the specified warnings as errors - #pragma warning ( error: 4800 )

// Function prototype
void PrintHelp(LPCTSTR pszAppName);

/*--------------------------------------------------------------------
The main function is the main routine. It parses the command-line
arguments and either removes or adds a drive letter.
Parameters:
   argc -  Count of the command-line arguments
   argv -  Array of pointers to the individual command-line arguments
--------------------------------------------------------------------*/
void wmain(int argc, WCHAR *argv[])
{
   WCHAR * pszDriveLetter, * pszNTDevice, * pszOptions;
   WCHAR szUniqueVolumeName[MAX_PATH];
   WCHAR szDriveLetterAndSlash[4];
   WCHAR szDriveLetter[3];
   BOOL  fRemoveDriveLetter;
   BOOL  fResult;

   if (argc != 3)
   {
      PrintHelp(argv[0]);
      return;
   }

   // Use the command line to see if user wants to add or remove the
   // drive letter. Do this by looking for the -r option.
   fRemoveDriveLetter = !lstrcmpi (argv[1], L"-r");

   if (fRemoveDriveLetter)
   {
      // User wants to remove the drive letter. Command line should
      // be: dl -r <drive letter>
      pszOptions       = argv[1];
      pszDriveLetter   = argv[2];
      pszNTDevice      = NULL;
   }
```

```
   else
   {
      // User wants to add a drive letter. Command line should be:
      // dl <drive letter> <NT device name>
      pszOptions       = NULL;
      pszDriveLetter   = argv[1];
      pszNTDevice      = argv[2];
   }

   // GetVolumeNameForVolumeMountPoint, SetVolumeMountPoint, and
   // DeleteVolumeMountPoint require drive letters to have a trailing
   // backslash. However, DefineDosDevice requires that the trailing
   // backslash be absent. So, use:
   //
   //    szDriveLetterAndSlash     for the mounted folder functions
   //    szDriveLetter             for DefineDosDevice
   //
   // This way, command lines that use a: or a:\
   // for drive letters can be accepted without writing back
   // to the original command-line argument.
   szDriveLetter[0] = pszDriveLetter[0];
   szDriveLetter[1] = ':';
   szDriveLetter[2] = '\0';

   szDriveLetterAndSlash[0] = pszDriveLetter[0];
   szDriveLetterAndSlash[1] = ':';
   szDriveLetterAndSlash[2] = '\\';
   szDriveLetterAndSlash[3] = '\0';

   // Now add or remove the drive letter.
   if (fRemoveDriveLetter)
   {
      fResult = DeleteVolumeMountPoint (szDriveLetterAndSlash);

      if (!fResult)
         wprintf(L"error %lu: couldn't remove %s\n", GetLastError(),
szDriveLetterAndSlash);
   }
   else
   {
      // To add a drive letter that persists through reboots, use
      // SetVolumeMountPoint. This requires the volume GUID path
      // of the device to which the new drive letter will refer.
      // To get the volume GUID path, use
      // GetVolumeNameForVolumeMountPoint; it requires the drive
      // letter to already exist. So, first define the drive
      // letter as a symbolic link to the device name. After
      // you have the volume GUID path the new drive letter will
      // point to, you must delete the symbolic link because the
      // mount manager allows only one reference to a device at a
      // time (the new one to be added).
      fResult = DefineDosDevice (DDD_RAW_TARGET_PATH, szDriveLetter,
pszNTDevice);

      if (fResult)
      {
```

```
            // If GetVolumeNameForVolumeMountPoint fails, then
            // SetVolumeMountPoint will also fail. However,
            // DefineDosDevice must be called to remove the temporary symbolic
link.
            // Therefore, set szUniqueVolume to a known empty string.
         if (!GetVolumeNameForVolumeMountPoint (szDriveLetterAndSlash,
szUniqueVolumeName, MAX_PATH))
         {
            DEBUG_PRINT("GetVolumeNameForVolumeMountPoint failed",
GetLastError());
            szUniqueVolumeName[0] = '\0';
         }

         fResult = DefineDosDevice (
                     DDD_RAW_TARGET_PATH|DDD_REMOVE_DEFINITION|
                     DDD_EXACT_MATCH_ON_REMOVE, szDriveLetter,
                     pszNTDevice);

         if (!fResult)
            DEBUG_PRINT("DefineDosDevice failed", GetLastError());

         fResult = SetVolumeMountPoint (szDriveLetterAndSlash,
szUniqueVolumeName);

         if (!fResult)
            wprintf(L"error %lu: could not add %s\n", GetLastError(),
szDriveLetterAndSlash);
      }
   }
}

/*--------------------------------------------------------------------
The PrintHelp function prints the command-line usage help.
Parameters: pszAppName
      The name of the executable. Used in displaying the help.
--------------------------------------------------------------------*/
void PrintHelp (LPCTSTR pszAppName)
{
   wprintf(L"---Adds/removes a drive letter assignment for a device---\n\n");
   wprintf(L"Usage: %s <Drive> <Device name> add a drive letter\n", pszAppName);
   wprintf(L"       %s -r <Drive>            remove a drive letter\n\n",
pszAppName);
   wprintf(L"Example: %s e:\\ \\Device\\CdRom0\n", pszAppName);
   wprintf(L"         %s -r e:\\\n", pszAppName);
}

#if defined (DEBUG)
/*--------------------------------------------------------------------
The DebugPrint function prints a string to STDOUT.

Parameters
   pszMsg
      The string to be printed to STDOUT.
   dwErr
      The error code; usually obtained from GetLastError. If dwErr is
      zero, no error code is added to the error string. If dwErr is
```

```
      nonzero, the error code will be printed in the error string.
-----------------------------------------------------------------------*/
void DebugPrint (LPCTSTR pszMsg, DWORD dwErr)
{
   if (dwErr)
      wprintf(L"%s: %lu\n", pszMsg, dwErr);
   else
      wprintf(L"%s\n", pszMsg);
}
#endif
```

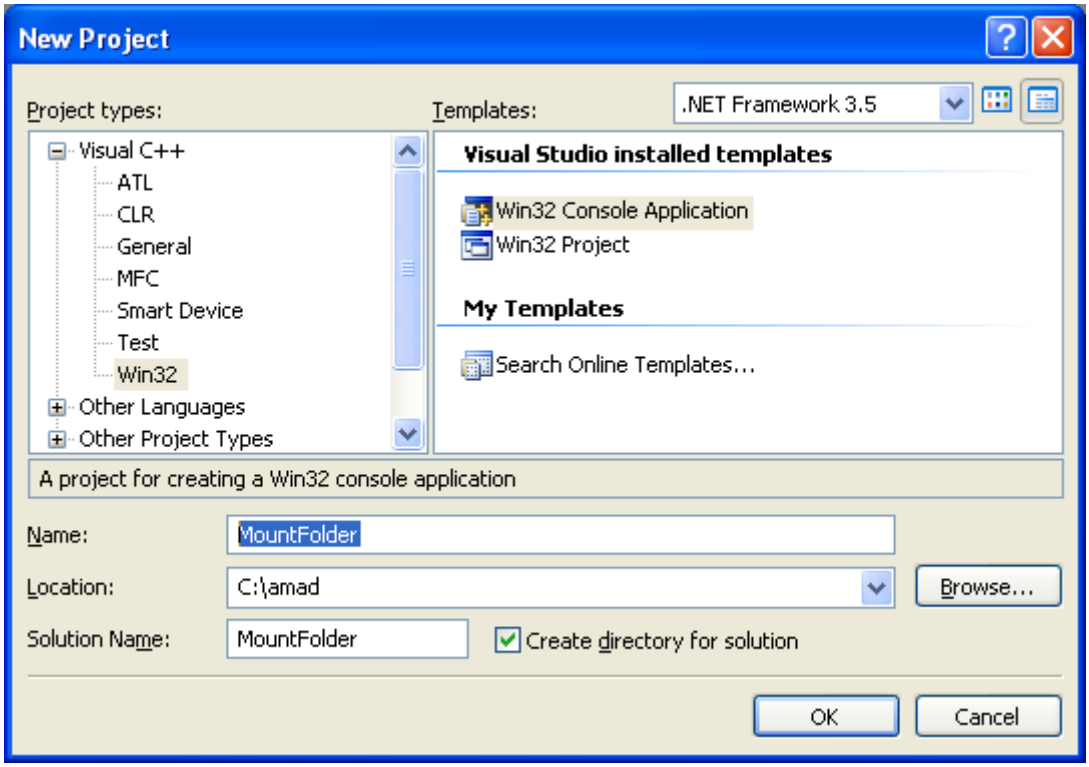
Build and run the project. The following screenshot is an output sample. This sample program should not be tested on the production machine.



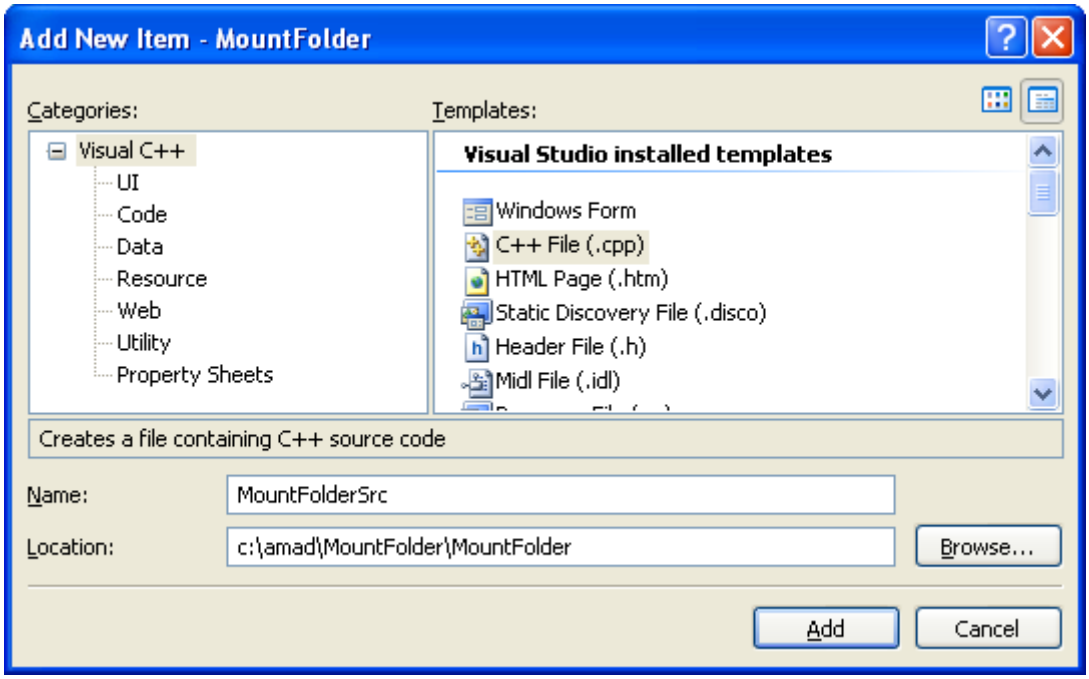
Creating a Mounted Folder Program Example

From the user point of view we can mount a drive/volume/partition to an NTFS empty folder as stated in the following steps. This will allow you to use a hard-drive as a normal folder on your main system drive and will be useful in many cases in managing your local storage. For example if you have made your original system drive/partition too small for your growing software needs or if you just want to be able to access multiple drives via one single drive letter.

The following sample demonstrates how to create a mounted folder programmatically. This sample uses the following functions: GetVolumeNameForVolumeMountPoint() and SetVolumeMountPoint().
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.

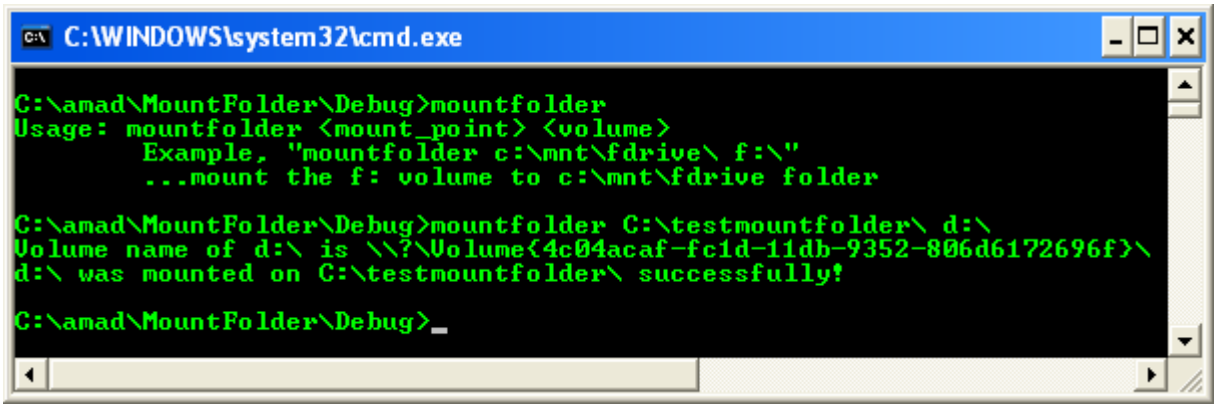


Add the following source code.

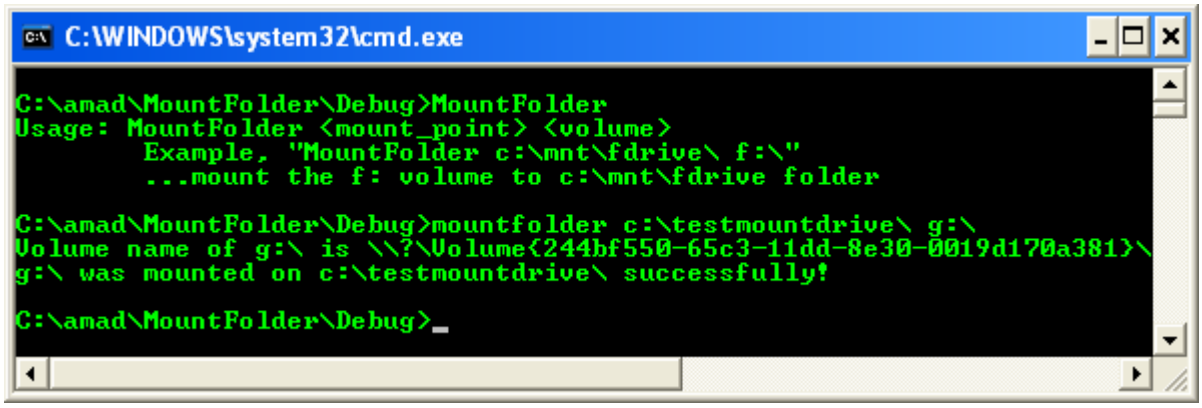
```
#include <windows.h>
#include <stdio.h>
```

```
// The following #define already defined in the SDK
// For Windows Server 2003, Windows XP
// #define _WIN32_WINNT 0x0501

#define BUFSIZE MAX_PATH

int wmain(int argc, WCHAR *argv[])
{
   BOOL bFlag;
   WCHAR Buf[BUFSIZE];     // temporary buffer for volume name

   if(argc != 3)
   {
      wprintf(L"Usage: %s <mount_point> <volume>\n", argv[0] );
      wprintf(L"  Example, \"%s c:\\mnt\\fdrive\\ f:\\\"\n", argv[0]);
        wprintf(L"      ...mount the f: volume to c:\\mnt\\fdrive folder\n");
      return(-1);
   }

   // We should do some error checking on the inputs. Make sure
   // there are colons and backslashes in the right places, etc.
   bFlag = GetVolumeNameForVolumeMountPoint(
              argv[2], // input volume mount point or directory
                  Buf, // output volume name buffer
              BUFSIZE  // size of volume name buffer
           );

   if (bFlag != TRUE)
   {
      wprintf(L"Retrieving volume name for %s failed.\n", argv[2]);
      return (-2);
   }

   wprintf(L"Volume name of %s is %s\n", argv[2], Buf);
   bFlag = SetVolumeMountPoint(
              argv[1], // mount point
                  Buf  // volume to be mounted
           );

   if (!bFlag)
   {
         wprintf(L"Attempt to mount %s at %s failed.\n", argv[2], argv[1]);
         wprintf(L"Error code is %d\n", GetLastError());
   }
   else
         wprintf(L"%s was mounted on %s successfully!\n", argv[2], argv[1]);

   return (bFlag);
}
```

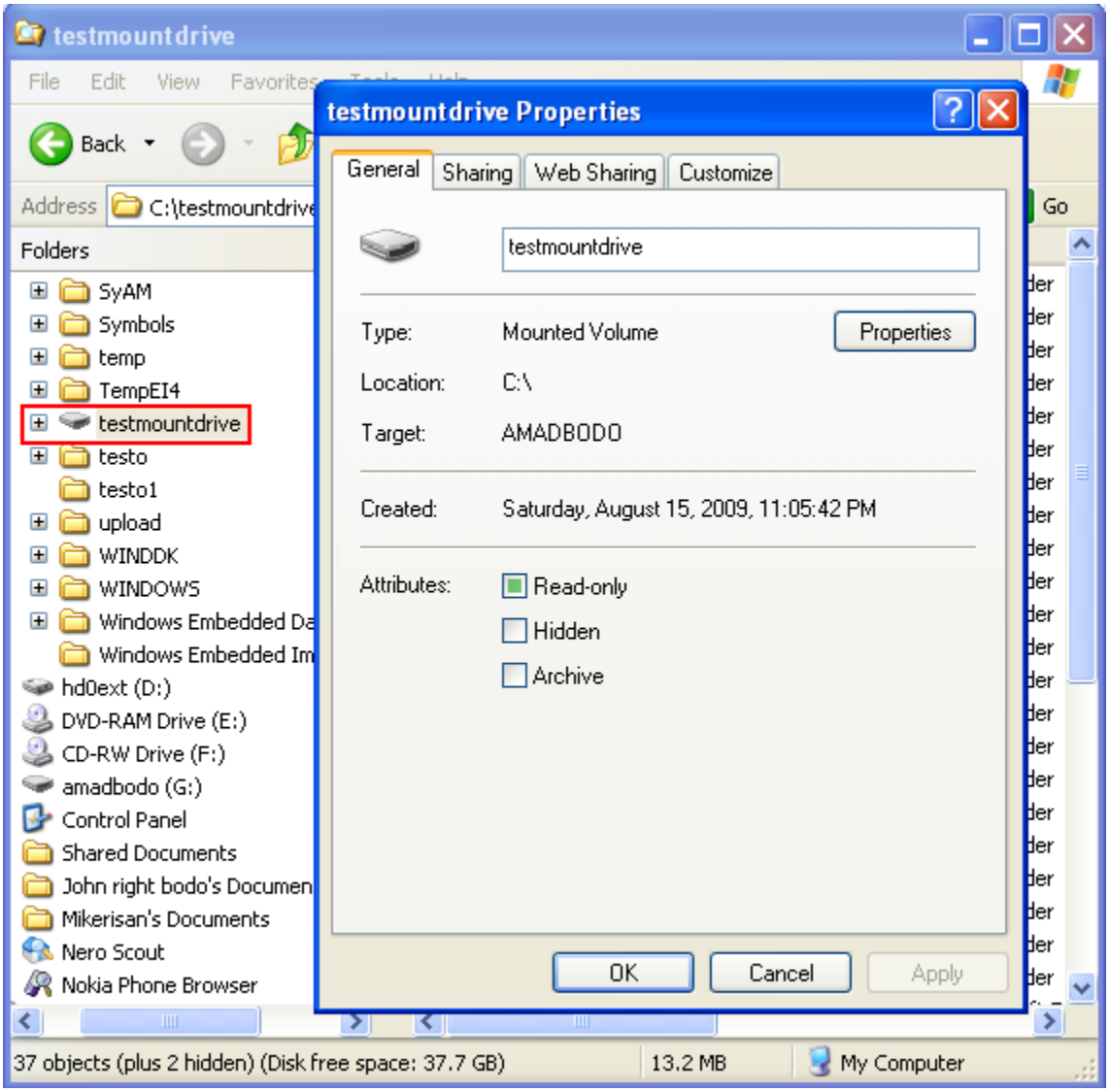
Build and run the project. The following screenshot is an output sample.



The following sample output shows a thumb drive (G:) was mounted on the C:\testmountdrive folder.

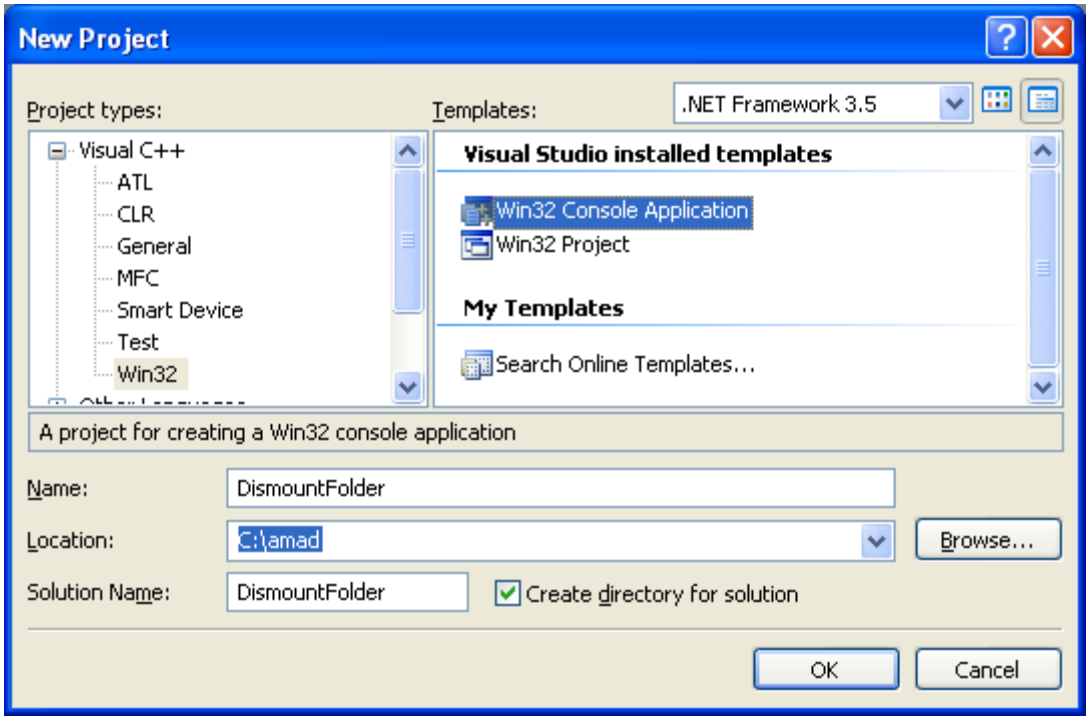


The following Figure shows the mounted drive on folder testmountdrive seen in the Windows explorer and the property page of the mounted drive.

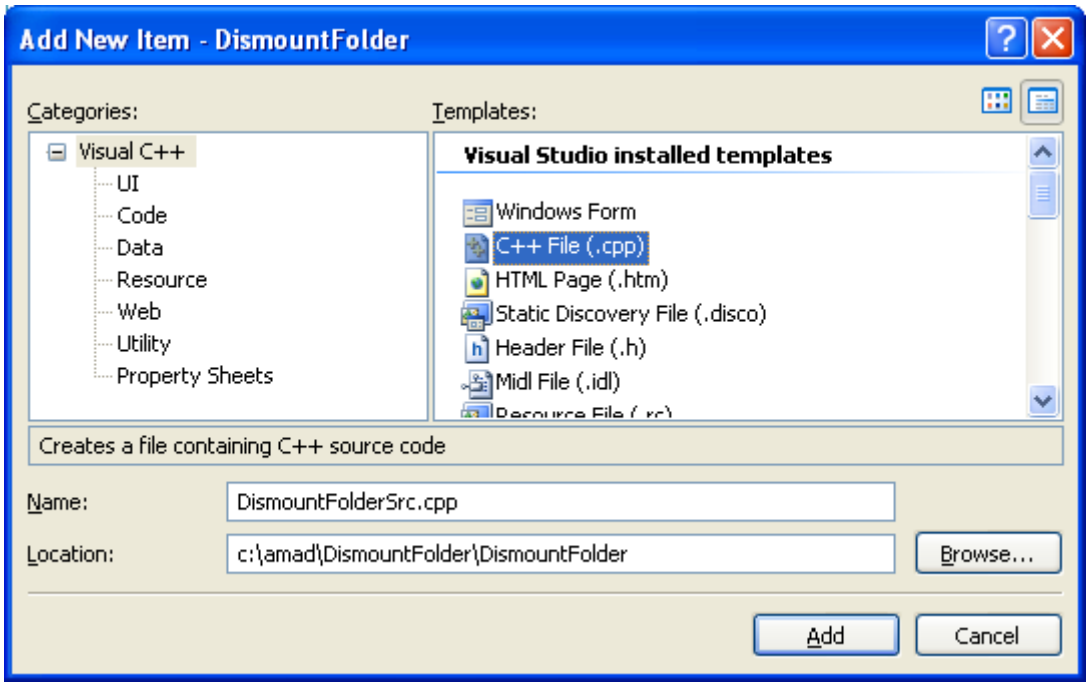


Deleting a Mounted Folder Program Example

The following program example shows you how to delete a mounted folder by using the DeleteVolumeMountPoint() function.
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.

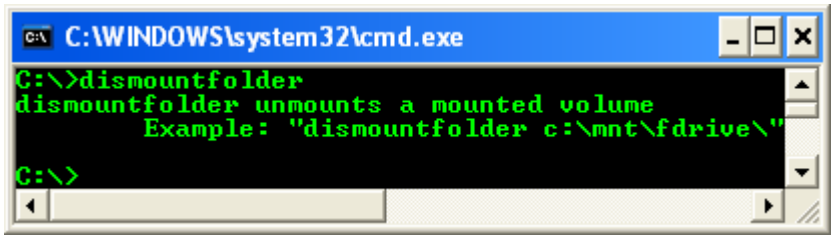


Add the following source code.

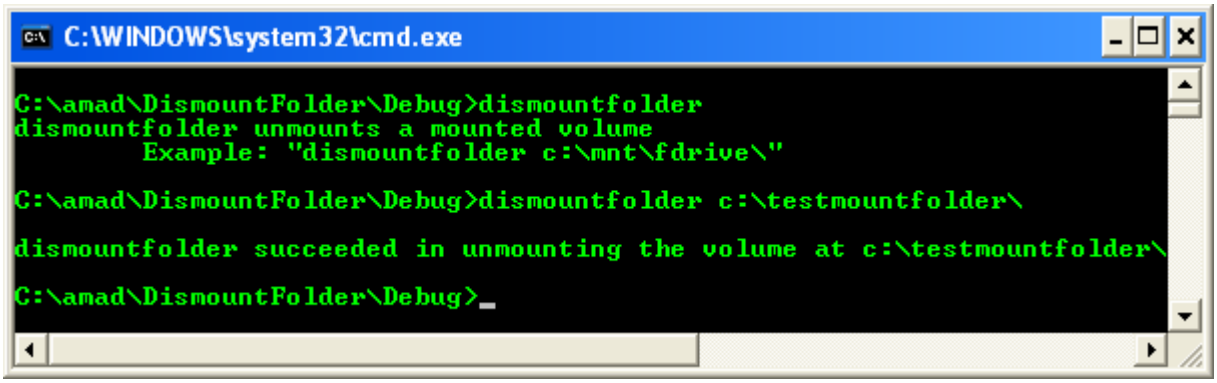
```
#include <windows.h>
#include <stdio.h>
```

```
void Syntax(WCHAR *argv)
{
   wprintf(L"%s unmounts a mounted volume\n", argv);
   wprintf(L"     Example: \"%s c:\\mnt\\fdrive\\\"\n", argv);
}

int wmain(int argc, WCHAR *argv[])
{
   BOOL bFlag;

   // Verify the arguments
   if (argc != 2)
   {
      Syntax(argv[0]);
      return (-1);
   }

   // We should do some error checking on the path argument, such as
   // ensuring that there is a trailing backslash
   bFlag = DeleteVolumeMountPoint(
              argv[1] // Path of the volume mount point
           );

   wprintf(L"\n%s %s in unmounting the volume at %s\n", argv[0],
           bFlag ? L"succeeded" : L"failed", argv[1]);

   return (bFlag);
}
```

Build and run the project. The following screenshot is an output sample without any argument.



The following sample outputs show unmounting the previous mounted drive.



Windows Master File Table (MFT)

The NTFS file system contains a file called the **master file table**, or MFT. There is at least one entry in the MFT for every file on an NTFS file system volume, including the MFT itself. All information about a file, including its size, time and date stamps, permissions, and data content, is stored either in MFT entries, or in space outside the MFT that is described by MFT entries.
As files are added to an NTFS file system volume, more entries are added to the MFT and the MFT increases in size. When files are deleted from an NTFS file system volume, their MFT entries are marked as free and may be reused. However, disk space that has been allocated for these entries is not reallocated, and the size of the MFT does not decrease.
Because utilities that defragment NTFS file system volumes on Windows 2000 cannot move MFT entries, and because excessive fragmentation of the MFT can impact performance, the NTFS file system reserves space for the MFT to keep the MFT as contiguous as possible as it grows. The space reserved by the NTFS file system for the MFT in each volume is called the MFT zone. Space for file and directories are also allocated from this space, but only after all of the volume space outside of the MFT zone has been allocated.
Depending on the average file size and other variables, either the reserved MFT zone or the unreserved space on the disk may be allocated first as the disk fills to capacity. Volumes with a small number of relatively large files will allocate the unreserved space first, while volumes with a large number of relatively small files allocate the MFT zone first. In either case, fragmentation of the MFT starts to take place when one region or the other becomes fully allocated. If the unreserved space is completely allocated, space for user files and directories will be allocated from the MFT zone. If the MFT zone is completely allocated, space for new MFT entries will be allocated from the unreserved space.
The MFT could not be used for defragmentation under Windows 2000, but this restriction is removed in Windows XP and later. Also, the MFT itself can be defragmented. To reduce the chance of the MFT zone becoming fully allocated before the defragmentation process is complete, leave as much space at the beginning of the MFT zone as possible before defragmenting the volume. If the MFT zone becomes fully allocated before defragmentation has completed, there must be unallocated space outside of the MFT zone.
The default MFT zone is calculated and reserved by the system when it mounts the volume, and is based on volume size. You can increase the MFT zone by means of the registry entry detailed in Microsoft Knowledge Base Article 174619, but you cannot make the default MFT zone smaller than what is calculated. Increasing the MFT zone does not decrease the disk space that users can use for data files.
To determine the current size of the MFT, analyze the NTFS file system drive with Disk Defragmenter, then click the **View Report** button. The drive statistics will be displayed, including the current MFT size, and number of fragments. You can also obtain the size of the MFT by using the FSCTL_GET_NTFS_VOLUME_DATA control code.
The master file table (MFT) stores the information required to retrieve files from an NTFS partition. A file may have one or more MFT records, and can contain one or more attributes. In NTFS, a file reference is the MFT segment reference of the base file record. The MFT contains file record segments; the first 16 of these are reserved for special files, such as the following:

1. 0: MFT ($Mft)
2. 5: root directory (\)

3. 6: volume cluster allocation file ($Bitmap)
4. 8: bad-cluster file ($BadClus)

Each file record segment starts with a file record segment header. For more information, see FILE_RECORD_SEGMENT_HEADER. Each file record segment is followed by one or more attributes. Each attribute starts with an attribute record header. For more information, see ATTRIBUTE_RECORD_HEADER. The attribute record includes the attribute type (such as $DATA or $BITMAP), an optional name, and the attribute value. The user data stream is an attribute, as are all streams. The attribute list is terminated with 0xFFFFFFFF ($END).
The following are some example attributes.

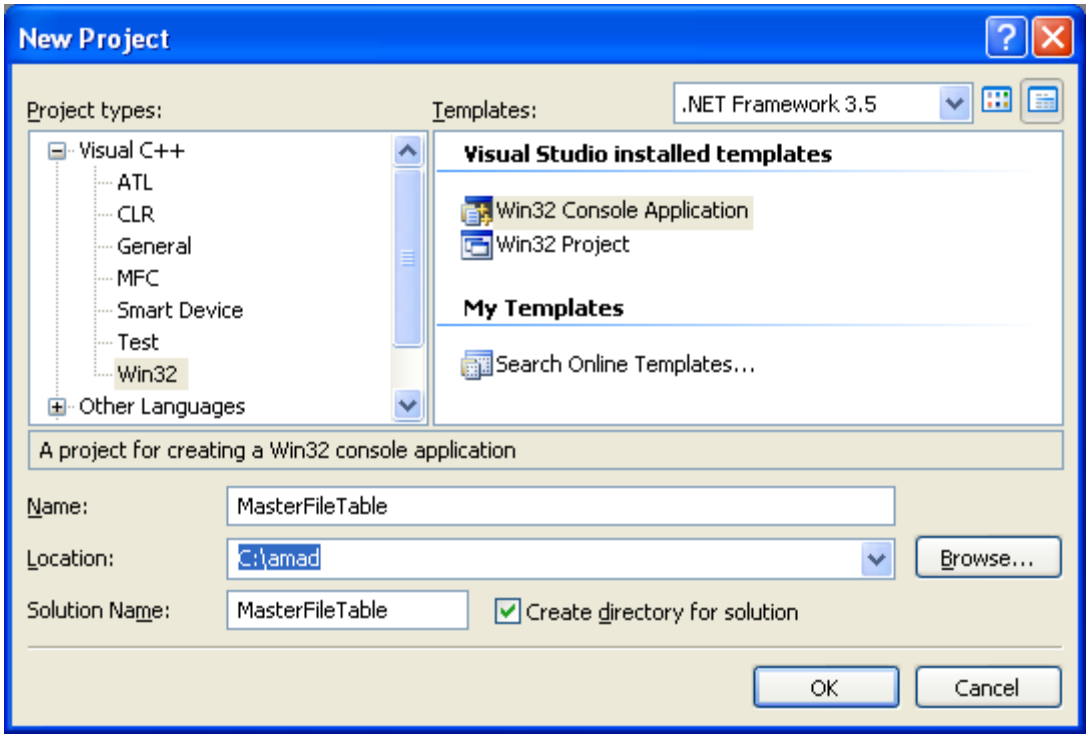
1. The $Mft file contains an unnamed $DATA attribute that is the sequence of MFT record segments, in order.
2. The $Mft file contains an unnamed $BITMAP attribute that indicates which MFT records are in use.
3. The $Bitmap file contains an unnamed $DATA attribute that indicates which clusters are in use.
4. The $BadClus file contains a $DATA attribute named $BAD that contains an entry that corresponds to each bad cluster.

When there is no more space for storing attributes in the file record segment, additional file record segments are allocated and inserted in the first (or base) file record segment in an attribute called the attribute list. The attribute list indicates where each attribute associated with the file can be found. This includes all attributes in the base file record, except for the attribute list itself. Structures related to the MFT include the following:

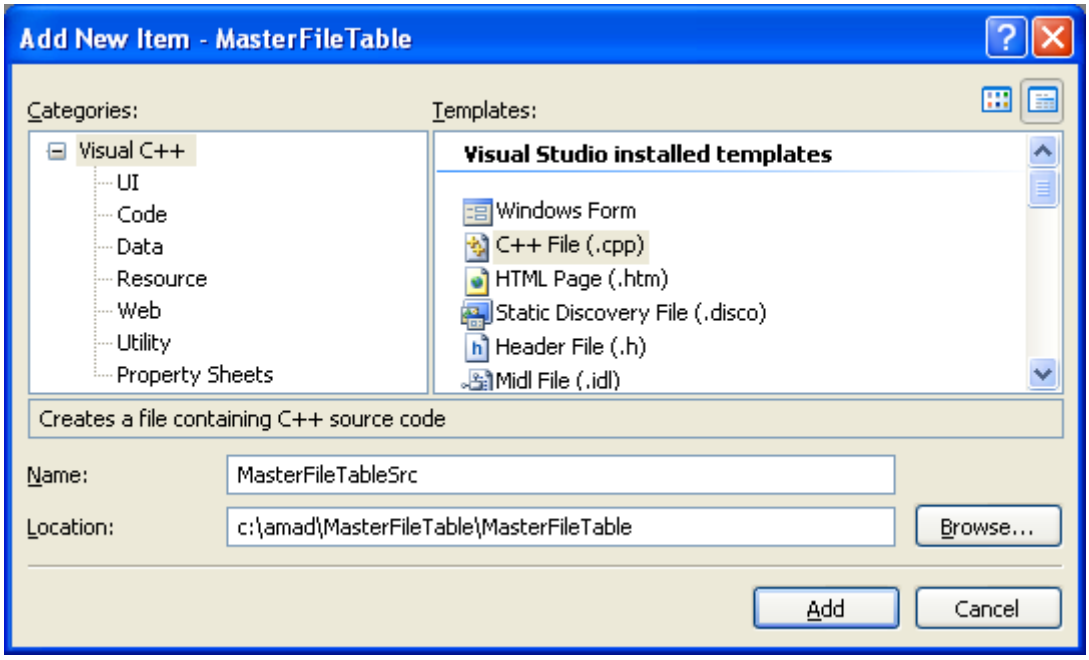
1. ATTRIBUTE_LIST_ENTRY
2. ATTRIBUTE_RECORD_HEADER
3. FILE_NAME
4. FILE_RECORD_SEGMENT_HEADER
5. MFT_SEGMENT_REFERENCE
6. MULTI_SECTOR_HEADER
7. STANDARD_INFORMATION

Master File Table Program Example 1

The following program example tries to read the Master File Table and extract some of the information. Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

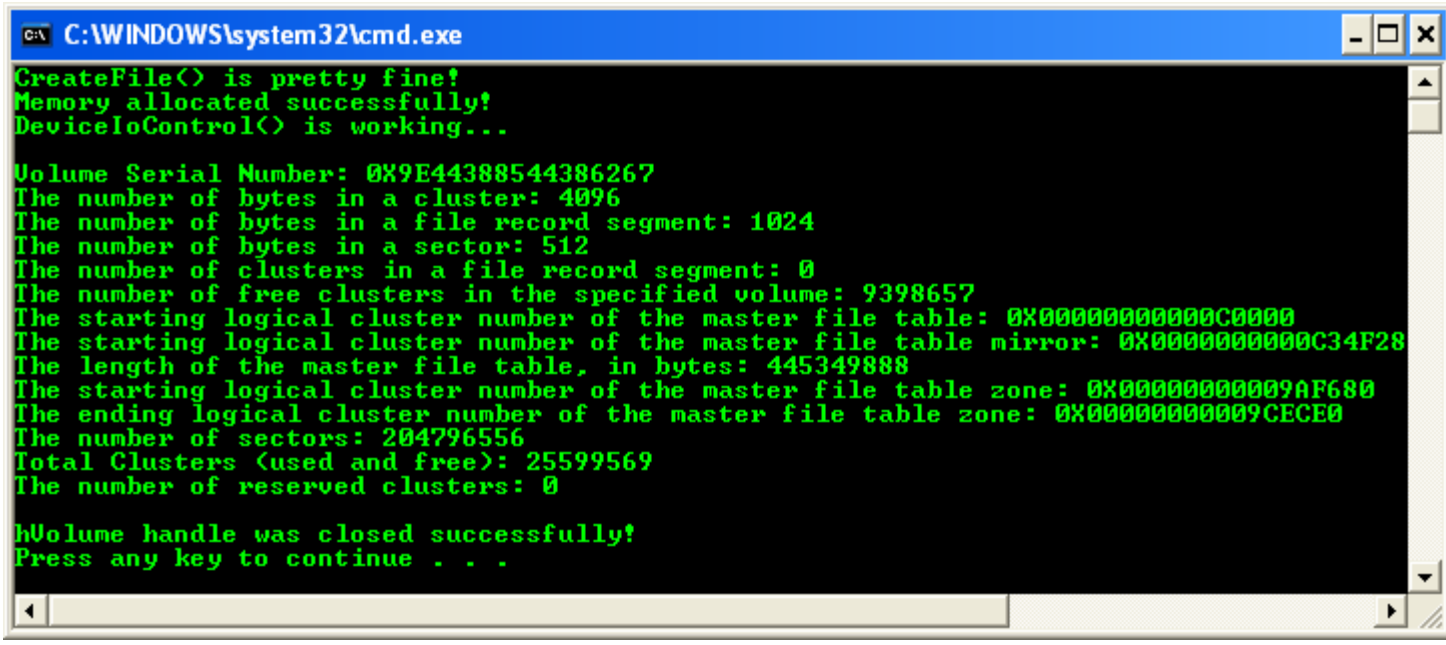
```
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>
```

```
// Format the Win32 system error code to string
void ErrorMessage(DWORD dwCode);

int wmain(int argc, WCHAR **argv)
{
	HANDLE hVolume;
	LPWSTR lpDrive = L"\\\\.\\c:";
	// {0} ~ ZeroMemory()
	PNTFS_VOLUME_DATA_BUFFER ntfsVolData = {0};
	// NTFS_EXTENDED_VOLUME_DATA versionMajMin = {0};
	BOOL bDioControl = FALSE;
	DWORD dwWritten = 0;

	hVolume = CreateFile(lpDrive,
		GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ | FILE_SHARE_WRITE,
		NULL,
		OPEN_EXISTING,
		0,
		NULL);

	if(hVolume == INVALID_HANDLE_VALUE)
	{
		wprintf(L"CreateFile() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
			wprintf(L"hVolume handle was closed successfully!\n");
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
	else
		wprintf(L"CreateFile() is pretty fine!\n");

	ntfsVolData =
(PNTFS_VOLUME_DATA_BUFFER)malloc(sizeof(NTFS_VOLUME_DATA_BUFFER)+sizeof(NTFS_EXT
ENDED_VOLUME_DATA));

	if(ntfsVolData == NULL)
		wprintf(L"Insufficient memory!\n");
	else
		wprintf(L"Memory allocated successfully!\n");

	// a call to FSCTL_GET_NTFS_VOLUME_DATA returns the structure
NTFS_VOLUME_DATA_BUFFER
	bDioControl = DeviceIoControl(hVolume, FSCTL_GET_NTFS_VOLUME_DATA, NULL,
0, ntfsVolData,
		sizeof(NTFS_VOLUME_DATA_BUFFER)+sizeof(NTFS_EXTENDED_VOLUME_DATA),
&dwWritten, NULL);

	// Failed or pending
	if(bDioControl == 0)
```

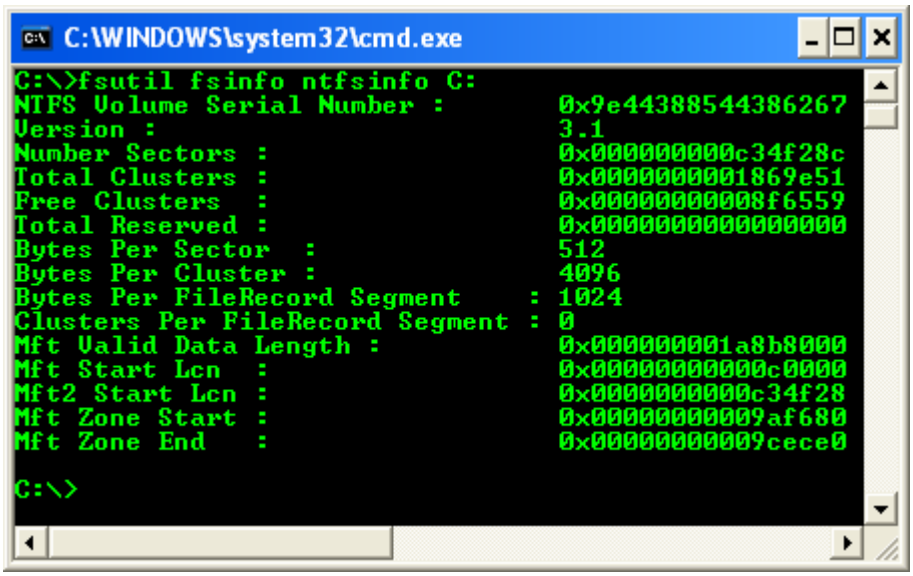
```
	{
		wprintf(L"DeviceIoControl() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
			wprintf(L"hVolume handle was closed successfully!\n");
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
	else
		wprintf(L"DeviceIoControl() is working...\n\n");

	wprintf(L"Volume Serial Number: 0X%.8X%.8X\n",ntfsVolData-
>VolumeSerialNumber.HighPart, ntfsVolData->VolumeSerialNumber.LowPart);
	wprintf(L"The number of bytes in a cluster: %u\n",ntfsVolData-
>BytesPerCluster);
	wprintf(L"The number of bytes in a file record segment: %u\n",ntfsVolData-
>BytesPerFileRecordSegment);
	wprintf(L"The number of bytes in a sector: %u\n",ntfsVolData-
>BytesPerSector);
	wprintf(L"The number of clusters in a file record segment:
%u\n",ntfsVolData->ClustersPerFileRecordSegment);
	wprintf(L"The number of free clusters in the specified volume:
%u\n",ntfsVolData->FreeClusters);
	wprintf(L"The starting logical cluster number of the master file table:
0X%.8X%.8X\n",ntfsVolData->MftStartLcn.HighPart,ntfsVolData-
>MftStartLcn.LowPart);
	wprintf(L"The starting logical cluster number of the master file table
mirror: 0X%.8X%.8X\n",ntfsVolData->Mft2StartLcn.HighPart, ntfsVolData-
>Mft2StartLcn.LowPart);
	wprintf(L"The length of the master file table, in bytes:
%u\n",ntfsVolData->MftValidDataLength);
	wprintf(L"The starting logical cluster number of the master file table
zone: 0X%.8X%.8X\n",ntfsVolData->MftZoneStart.HighPart,ntfsVolData-
>MftZoneStart.LowPart);
	wprintf(L"The ending logical cluster number of the master file table zone:
0X%.8X%.8X\n",ntfsVolData->MftZoneEnd.HighPart, ntfsVolData-
>MftZoneEnd.LowPart);
	wprintf(L"The number of sectors: %u\n",ntfsVolData->NumberSectors);
	wprintf(L"Total Clusters (used and free): %u\n",ntfsVolData-
>TotalClusters);
	wprintf(L"The number of reserved clusters: %u\n",ntfsVolData-
>TotalReserved);

	// To extract this info the buffer must be large enough, however...FAILED!
	//wprintf(L"Byte returns: %u\n", versionMajMin.ByteCount);
	//wprintf(L"Major version: %u\n", versionMajMin.MajorVersion);
	//wprintf(L"Minor version: %u\n", versionMajMin.MinorVersion);

	if(CloseHandle(hVolume) != 0)
		wprintf(L"\nhVolume handle was closed successfully!\n");
	else
	{
```

```
            wprintf(L"\nFailed to close hVolume handle!\n");
            ErrorMessage(GetLastError());
      }

      // free up the allocated memory by malloc()
      free(ntfsVolData);

      return 0;
}

// Accessory function converting the GetLastError() code
// to a meaningful string
void ErrorMessage(DWORD dwCode)
{
    // get the error code...
    DWORD dwErrCode = dwCode;
      DWORD dwNumChar;

    LPWSTR szErrString = NULL;  // will be allocated and filled by FormatMessage

    dwNumChar = FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM, // use windows internal message
table
                0,       // 0 since source is internal message table
                dwErrCode, // this is the error code number
                0,       // auto-determine language to use
                (LPWSTR)&szErrString, // the messsage
                0,                 // min size for buffer
                0 );               // since getting message from system tables
      if(dwNumChar == 0)
            wprintf(L"FormatMessage() failed, error %u\n", GetLastError());
      //else
      //    wprintf(L"FormatMessage() should be fine!\n");

     wprintf(L"Error code %u:\n  %s\n", dwErrCode, szErrString) ;

      // This buffer used by FormatMessage()
    if(LocalFree(szErrString) != NULL)
            wprintf(L"Failed to free up the buffer, error %u\n",
GetLastError());
      //else
      //    wprintf(L"Buffer has been freed\n");
  }
```

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

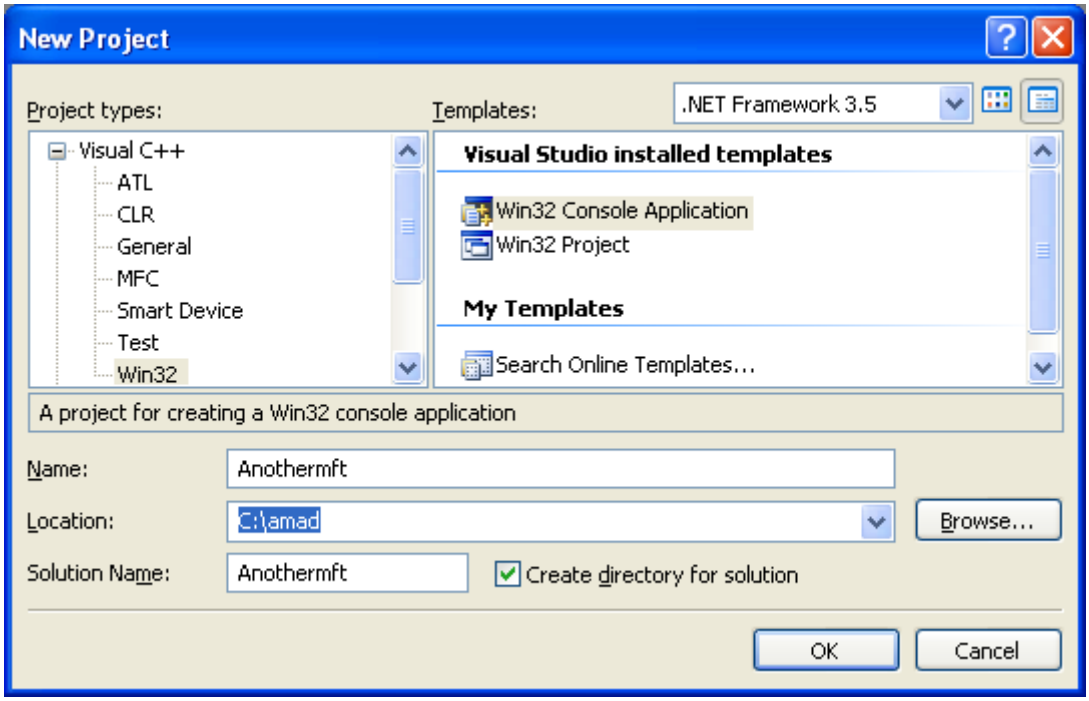


You can the result with the fsutil, the Windows file system utility.

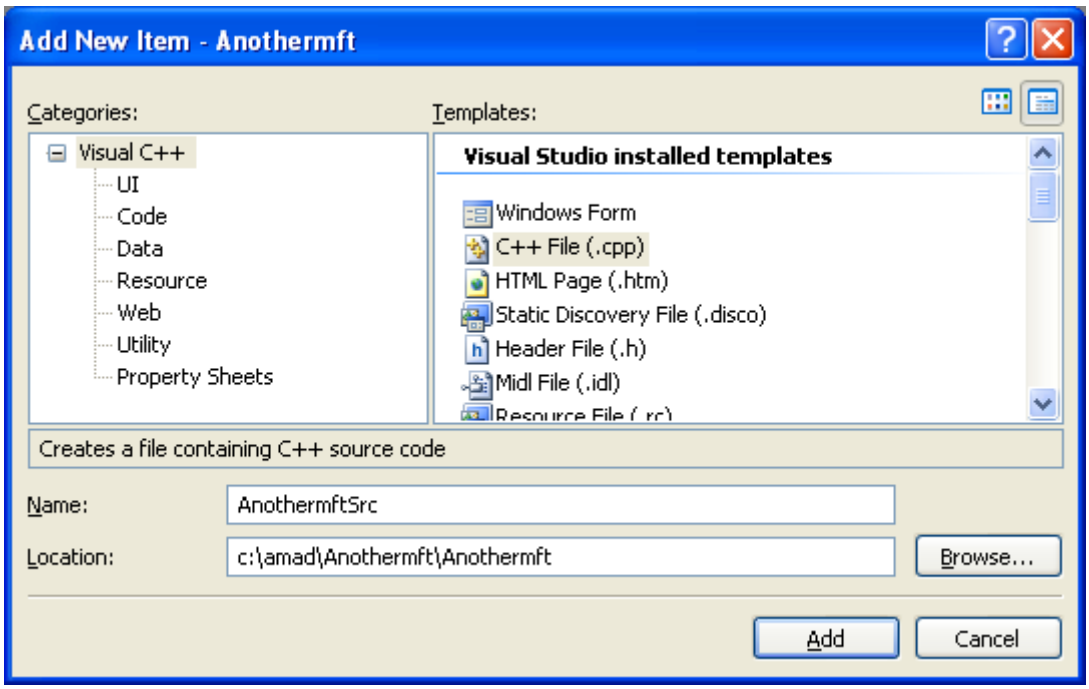


Master File Table Program Example 2: Reading and Dumping the Deleted Files

The following program example tries to read the file record header from Master File Table. Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

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

```
typedef struct {
      ULONG Type;
      USHORT UsaOffset;
      USHORT UsaCount;
      USN Usn;
} NTFS_RECORD_HEADER, *PNTFS_RECORD_HEADER;

// Type needed for interpreting the MFT-records
typedef struct {
    NTFS_RECORD_HEADER RecHdr;    // An NTFS_RECORD_HEADER structure with a Type
of 'FILE'.
    USHORT SequenceNumber;        // Sequence number - The number of times
                                  // that the MFT entry has been
reused.
    USHORT LinkCount;             // Hard link count - The number of directory
links to the MFT entry
    USHORT AttributeOffset;       // Offset to the first Attribute - The offset,
in bytes,
                                  // from the start of the
structure to the first attribute of the MFT
    USHORT Flags;                 // Flags - A bit array of flags specifying
properties of the MFT entry
                                  // InUse 0x0001 - The MFT
entry is in use
                                  // Directory 0x0002 - The MFT
entry represents a directory
    ULONG BytesInUse;             // Real size of the FILE record - The number
of bytes used by the MFT entry.
    ULONG BytesAllocated;         // Allocated size of the FILE record - The
number of bytes
                                  // allocated for the MFT entry
    ULONGLONG BaseFileRecord;     // reference to the base FILE record - If the
MFT entry contains
                                  // attributes that overflowed
a base MFT entry, this member
                                  // contains the file reference
number of the base entry;
                                  // otherwise, it contains zero
    USHORT NextAttributeNumber;   // Next Attribute Id - The number that will be
assigned to
                                  // the next attribute added to
the MFT entry.
    USHORT Pading;                // Align to 4 byte boundary (XP)
    ULONG MFTRecordNumber;        // Number of this MFT Record (XP)
    USHORT UpdateSeqNum;          //
} FILE_RECORD_HEADER, *PFILE_RECORD_HEADER;


// Convert the Win32 system error code to string
void ErrorMessage(DWORD dwCode);

int wmain(int argc, WCHAR **argv)
{
      HANDLE hVolume;
      LPWSTR lpDrive = L"\\\\.\\c:";
      NTFS_VOLUME_DATA_BUFFER ntfsVolData = {0};
```

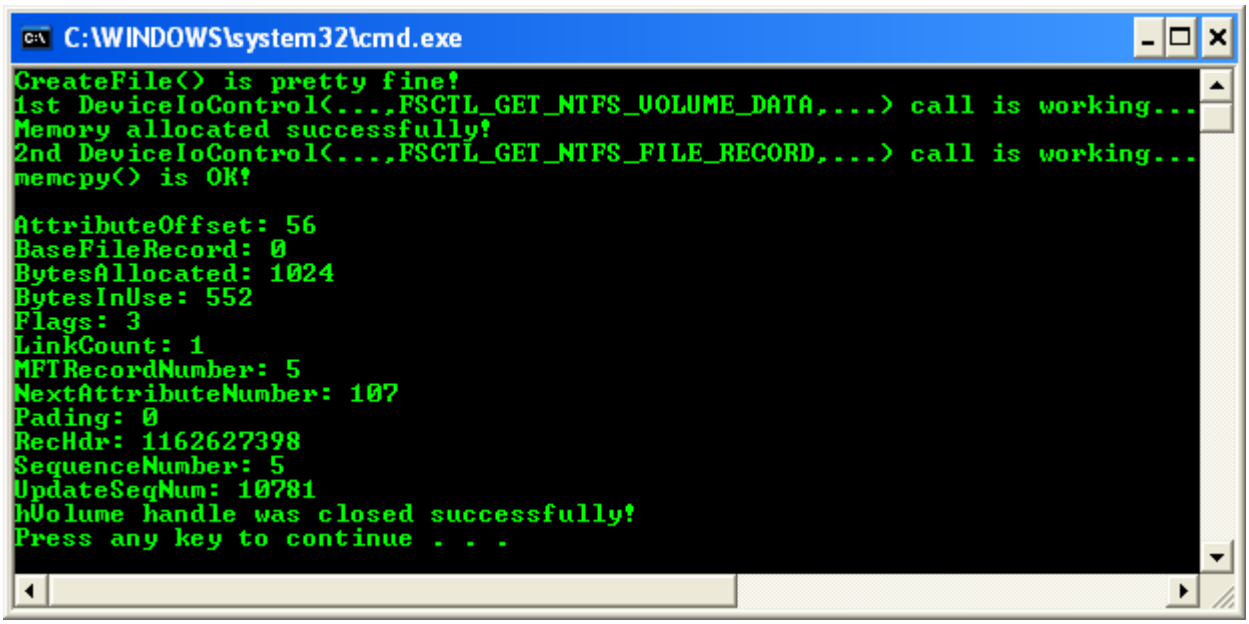
```
	BOOL bDioControl = FALSE;
	DWORD dwWritten = 0;
	DWORD lpBytesReturned = 0;
	FILE_RECORD_HEADER  FileRecHdr = {0};
	// Variables for MFT-reading
	NTFS_FILE_RECORD_INPUT_BUFFER   ntfsFileRecordInput;
	PNTFS_FILE_RECORD_OUTPUT_BUFFER ntfsFileRecordOutput;

	hVolume = CreateFile(lpDrive,
		GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ | FILE_SHARE_WRITE,
		NULL,
		OPEN_EXISTING,
		0,
		NULL);

	if(hVolume == INVALID_HANDLE_VALUE)
	{
		wprintf(L"CreateFile() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
			wprintf(L"hVolume handle was closed successfully!\n");
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
	else
		wprintf(L"CreateFile() is pretty fine!\n");

	// get ntfsVolData by calling DeviceIoControl()
	// with CtlCode FSCTL_GET_NTFS_VOLUME_DATA
	// setup output buffer - FSCTL_GET_NTFS_FILE_RECORD depends on this

	// a call to FSCTL_GET_NTFS_VOLUME_DATA returns the structure
NTFS_VOLUME_DATA_BUFFER
	bDioControl = DeviceIoControl(hVolume, FSCTL_GET_NTFS_VOLUME_DATA, NULL,
0, &ntfsVolData,
		sizeof(ntfsVolData), &dwWritten, NULL);

	// Failed or pending
	if(bDioControl == 0)
	{
		wprintf(L"DeviceIoControl() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
			wprintf(L"hVolume handle was closed successfully!\n");
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
```

```
	else
		wprintf(L"1st DeviceIoControl(...,FSCTL_GET_NTFS_VOLUME_DATA,...)
call is working...\n");

	//a call to FSCTL_GET_NTFS_VOLUME_DATA returns the structure
NTFS_VOLUME_DATA_BUFFER
	ntfsFileRecordOutput = (PNTFS_FILE_RECORD_OUTPUT_BUFFER)

	malloc(sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER)+ntfsVolData.BytesPerFileReco
rdSegment-1);

	if(ntfsFileRecordOutput == NULL)
		wprintf(L"Insufficient memory lol!\n");
	else
		wprintf(L"Memory allocated successfully!\n");

	// The MFT-record #5 is the root-dir???
	ntfsFileRecordInput.FileReferenceNumber.QuadPart = 5;

	bDioControl = DeviceIoControl(
		hVolume,
		FSCTL_GET_NTFS_FILE_RECORD,
	  &ntfsFileRecordInput,
	  sizeof(NTFS_FILE_RECORD_INPUT_BUFFER),
	  ntfsFileRecordOutput,

sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER)+ntfsVolData.BytesPerFileRecordSegment-1,
	  &lpBytesReturned, NULL);

	// Failed or pending
	if(bDioControl == 0)
	{
		wprintf(L"DeviceIoControl() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
		{
			wprintf(L"hVolume handle was closed successfully!\n");
		}
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
	else
		wprintf(L"2nd DeviceIoControl(...,FSCTL_GET_NTFS_FILE_RECORD,...)
call is working...\n");

		// read the record header from start of MFT-record
		if(!(memcpy(&FileRecHdr, &ntfsFileRecordOutput->FileRecordBuffer[0],
sizeof(FILE_RECORD_HEADER))))
			wprintf(L"memcpy() failed!\n");
		else
			wprintf(L"memcpy() is OK!\n\n");
```

```
            wprintf(L"AttributeOffset: %u\n",FileRecHdr.AttributeOffset);
            wprintf(L"BaseFileRecord: %u\n",FileRecHdr.BaseFileRecord);
            wprintf(L"BytesAllocated: %u\n",FileRecHdr.BytesAllocated);
            wprintf(L"BytesInUse: %u\n",FileRecHdr.BytesInUse);
            wprintf(L"Flags: %u\n",FileRecHdr.Flags);
            wprintf(L"LinkCount: %u\n",FileRecHdr.LinkCount);
            wprintf(L"MFTRecordNumber: %u\n",FileRecHdr.MFTRecordNumber);
            wprintf(L"NextAttributeNumber:
%u\n",FileRecHdr.NextAttributeNumber);
            wprintf(L"Pading: %u\n",FileRecHdr.Pading);
            wprintf(L"RecHdr: %u\n",FileRecHdr.RecHdr);
            wprintf(L"SequenceNumber: %u\n",FileRecHdr.SequenceNumber);
            wprintf(L"UpdateSeqNum: %u\n",FileRecHdr.UpdateSeqNum);

      if(CloseHandle(hVolume) != 0)
            wprintf(L"hVolume handle was closed successfully!\n");
      else
      {
            wprintf(L"Failed to close hVolume handle!\n");
            ErrorMessage(GetLastError());
      }

      // Free up the allocated memory by malloc()
      free(ntfsFileRecordOutput);

      return 0;
}

void ErrorMessage(DWORD dwCode)
{
    // get the error code...
    DWORD dwErrCode = dwCode;
    DWORD dwNumChar;

    LPWSTR szErrString = NULL;  // will be allocated and filled by FormatMessage

    dwNumChar = FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM, // use windows internal message
table
                0,       // 0 since source is internal message table
                dwErrCode, // this is the error code number
                0,       // auto-determine language to use
                (LPWSTR)&szErrString, // the messsage
                0,                 // min size for buffer
                0 );               // since getting message from system tables
      if(dwNumChar == 0)
            wprintf(L"FormatMessage() failed, error %u\n", GetLastError());
      //else
      //    wprintf(L"FormatMessage() should be fine!\n");

     wprintf(L"Error code %u:\n  %s\n", dwErrCode, szErrString) ;

      // This buffer used by FormatMessage()
    if(LocalFree(szErrString) != NULL)
            wprintf(L"Failed to free up the buffer, error %u\n",
GetLastError());
```

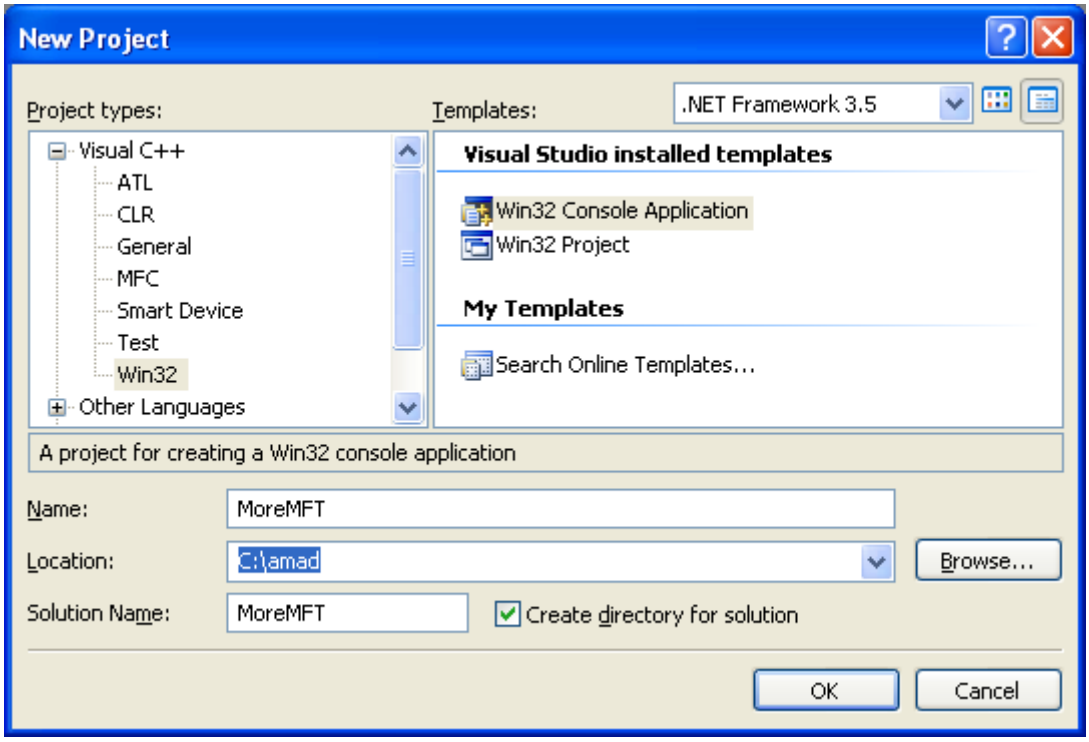
```
    //else
    //    wprintf(L"Buffer has been freed\n");
}
```

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

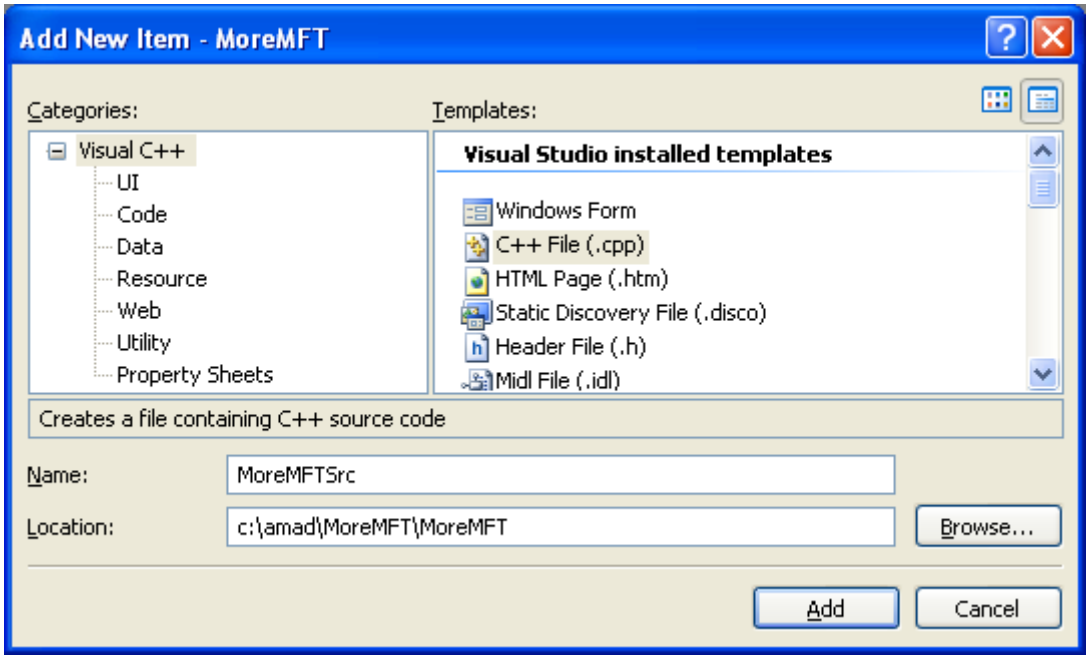


Master File Table Program Example 3: Using Non-Windows Types

The following program example tries to read the Master File Table using custom made types. Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

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

```
// These types should be stored in separate
// include file, not done here
typedef struct {
      ULONG Type;
      USHORT UsaOffset;
      USHORT UsaCount;
      USN Usn;
} NTFS_RECORD_HEADER, *PNTFS_RECORD_HEADER;

typedef struct {
      NTFS_RECORD_HEADER Ntfs;
      USHORT SequenceNumber;
      USHORT LinkCount;
      USHORT AttributesOffset;
      USHORT Flags; // 0x0001 = InUse, 0x0002 = Directory
      ULONG BytesInUse;
      ULONG BytesAllocated;
      ULONGLONG BaseFileRecord;
      USHORT NextAttributeNumber;
} FILE_RECORD_HEADER, *PFILE_RECORD_HEADER;

typedef enum {
      AttributeStandardInformation = 0x10,
      AttributeAttributeList = 0x20,
      AttributeFileName = 0x30,
      AttributeObjectId = 0x40,
      AttributeSecurityDescriptor = 0x50,
      AttributeVolumeName = 0x60,
      AttributeVolumeInformation = 0x70,
      AttributeData = 0x80,
      AttributeIndexRoot = 0x90,
      AttributeIndexAllocation = 0xA0,
      AttributeBitmap = 0xB0,
      AttributeReparsePoint = 0xC0,
      AttributeEAInformation = 0xD0,
      AttributeEA = 0xE0,
      AttributePropertySet = 0xF0,
      AttributeLoggedUtilityStream = 0x100
} ATTRIBUTE_TYPE, *PATTRIBUTE_TYPE;

typedef struct {
      ATTRIBUTE_TYPE AttributeType;
      ULONG Length;
      BOOLEAN Nonresident;
      UCHAR NameLength;
      USHORT NameOffset;
      USHORT Flags; // 0x0001 = Compressed
      USHORT AttributeNumber;
} ATTRIBUTE, *PATTRIBUTE;

typedef struct {
      ATTRIBUTE Attribute;
      ULONG ValueLength;
      USHORT ValueOffset;
      USHORT Flags; // 0x0001 = Indexed
```

```
} RESIDENT_ATTRIBUTE, *PRESIDENT_ATTRIBUTE;

typedef struct {
      ATTRIBUTE Attribute;
      ULONGLONG LowVcn;
      ULONGLONG HighVcn;
      USHORT RunArrayOffset;
      UCHAR CompressionUnit;
      UCHAR AlignmentOrReserved[5];
      ULONGLONG AllocatedSize;
      ULONGLONG DataSize;
      ULONGLONG InitializedSize;
      ULONGLONG CompressedSize; // Only when compressed
} NONRESIDENT_ATTRIBUTE, *PNONRESIDENT_ATTRIBUTE;

typedef struct {
      ULONGLONG CreationTime;
      ULONGLONG ChangeTime;
      ULONGLONG LastWriteTime;
      ULONGLONG LastAccessTime;
      ULONG FileAttributes;
      ULONG AlignmentOrReservedOrUnknown[3];
      ULONG QuotaId; // NTFS 3.0 only
      ULONG SecurityId; // NTFS 3.0 only
      ULONGLONG QuotaCharge; // NTFS 3.0 only
      USN Usn; // NTFS 3.0 only
} STANDARD_INFORMATION, *PSTANDARD_INFORMATION;

typedef struct {
      ATTRIBUTE_TYPE AttributeType;
      USHORT Length;
      UCHAR NameLength;
      UCHAR NameOffset;
      ULONGLONG LowVcn;
      ULONGLONG FileReferenceNumber;
      USHORT AttributeNumber;
      USHORT AlignmentOrReserved[3];
} ATTRIBUTE_LIST, *PATTRIBUTE_LIST;

typedef struct {
      ULONGLONG DirectoryFileReferenceNumber;
      ULONGLONG CreationTime; // Saved when filename last changed
      ULONGLONG ChangeTime; // ditto
      ULONGLONG LastWriteTime; // ditto
      ULONGLONG LastAccessTime; // ditto
      ULONGLONG AllocatedSize; // ditto
      ULONGLONG DataSize; // ditto
      ULONG FileAttributes; // ditto
      ULONG AlignmentOrReserved;
      UCHAR NameLength;
      UCHAR NameType; // 0x01 = Long, 0x02 = Short
      WCHAR Name[1];
} FILENAME_ATTRIBUTE, *PFILENAME_ATTRIBUTE;

// Format the Win32 system error code to string
void ErrorMessage(DWORD dwCode);
```

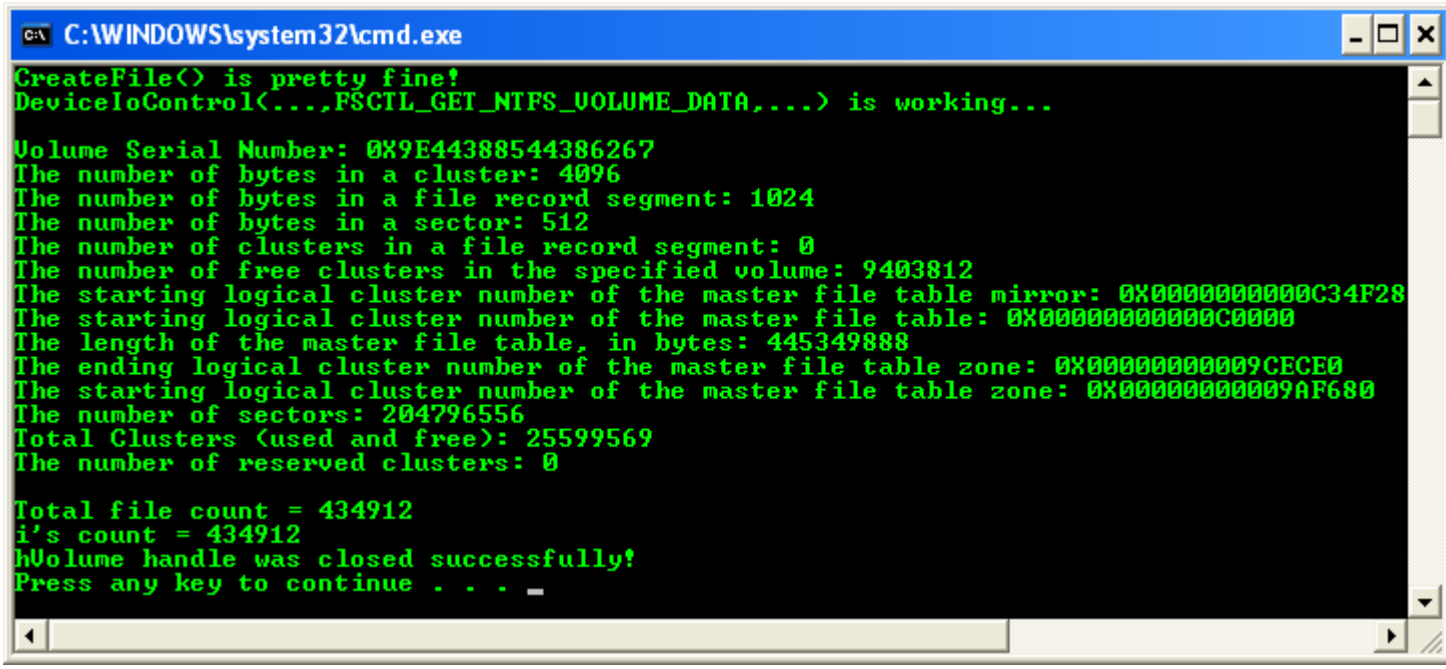
```
int wmain(int argc, WCHAR **argv)
{
	HANDLE hVolume;
	LPWSTR lpDrive = L"\\\\.\\c:";
	NTFS_VOLUME_DATA_BUFFER ntfsVolData = {0};
	BOOL bDioControl = FALSE;
	DWORD dwWritten = 0;
	LARGE_INTEGER num;
	LONGLONG total_file_count, i;
	NTFS_FILE_RECORD_INPUT_BUFFER mftRecordInput;
	PNTFS_FILE_RECORD_OUTPUT_BUFFER output_buffer;

	hVolume = CreateFile(lpDrive,
		GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ | FILE_SHARE_WRITE,
		NULL,
		OPEN_EXISTING,
		0,
		NULL);

	if(hVolume == INVALID_HANDLE_VALUE)
	{
		wprintf(L"CreateFile() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
			wprintf(L"hVolume handle was closed successfully!\n");
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
	else
		wprintf(L"CreateFile() is pretty fine!\n");

	// a call to FSCTL_GET_NTFS_VOLUME_DATA returns the structure
NTFS_VOLUME_DATA_BUFFER
	bDioControl = DeviceIoControl(hVolume, FSCTL_GET_NTFS_VOLUME_DATA, NULL,
0, &ntfsVolData,
		sizeof(NTFS_VOLUME_DATA_BUFFER), &dwWritten, NULL);

	if(bDioControl == 0)
	{
		wprintf(L"DeviceIoControl() failed!\n");
		ErrorMessage(GetLastError());
		if(CloseHandle(hVolume) != 0)
			wprintf(L"hVolume handle was closed successfully!\n");
		else
		{
			wprintf(L"Failed to close hVolume handle!\n");
			ErrorMessage(GetLastError());
		}
		exit(1);
	}
```

```
	else
		wprintf(L"DeviceIoControl(...,FSCTL_GET_NTFS_VOLUME_DATA,...) is
working...\n\n");

	wprintf(L"Volume Serial Number:
0X%.8X%.8X\n",ntfsVolData.VolumeSerialNumber.HighPart,ntfsVolData.VolumeSerialNu
mber.LowPart);
	wprintf(L"The number of bytes in a cluster:
%u\n",ntfsVolData.BytesPerCluster);
	wprintf(L"The number of bytes in a file record segment:
%u\n",ntfsVolData.BytesPerFileRecordSegment);
	wprintf(L"The number of bytes in a sector:
%u\n",ntfsVolData.BytesPerSector);
	wprintf(L"The number of clusters in a file record segment:
%u\n",ntfsVolData.ClustersPerFileRecordSegment);
	wprintf(L"The number of free clusters in the specified volume:
%u\n",ntfsVolData.FreeClusters);
	wprintf(L"The starting logical cluster number of the master file table
mirror: 0X%.8X%.8X\n",ntfsVolData.Mft2StartLcn.HighPart,
		ntfsVolData.Mft2StartLcn.LowPart);
	wprintf(L"The starting logical cluster number of the master file table:
0X%.8X%.8X\n",ntfsVolData.MftStartLcn.HighPart,
		ntfsVolData.MftStartLcn.LowPart);
	wprintf(L"The length of the master file table, in bytes:
%u\n",ntfsVolData.MftValidDataLength);
	wprintf(L"The ending logical cluster number of the master file table zone:
0X%.8X%.8X\n",ntfsVolData.MftZoneEnd.HighPart,
		ntfsVolData.MftZoneEnd.LowPart);
	wprintf(L"The starting logical cluster number of the master file table
zone: 0X%.8X%.8X\n",ntfsVolData.MftZoneStart.HighPart,
		ntfsVolData.MftZoneStart.LowPart);
	wprintf(L"The number of sectors: %u\n",ntfsVolData.NumberSectors);
	wprintf(L"Total Clusters (used and free):
%u\n",ntfsVolData.TotalClusters);
	wprintf(L"The number of reserved clusters:
%u\n\n",ntfsVolData.TotalReserved);

	num.QuadPart = 1024; // 1024 or 2048

	// We divide the MftValidDataLength (Master file table length) by 1024 to
find
	// the total entry count for the MFT
	total_file_count = (ntfsVolData.MftValidDataLength.QuadPart/num.QuadPart);

	wprintf(L"Total file count = %u\n", total_file_count);

	for(i = 0; i < total_file_count;i++)
	{
		mftRecordInput.FileReferenceNumber.QuadPart = i;

		// prior to calling the DeviceIoControl() we need to load
		// an input record with which entry number we want

		// setup outputbuffer - FSCTL_GET_NTFS_FILE_RECORD depends on this
```

```
            output_buffer =
(PNTFS_FILE_RECORD_OUTPUT_BUFFER)malloc(sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER)+n
tfsVolData.BytesPerFileRecordSegment-1);

            if(output_buffer == NULL)
            {
                  wprintf(L"malloc() failed - insufficient memory!\n");
                  ErrorMessage(GetLastError());
                  exit(1);
            }

            bDioControl = DeviceIoControl(hVolume,
FSCTL_GET_NTFS_FILE_RECORD,&mftRecordInput,
                  sizeof(mftRecordInput), output_buffer,
                  sizeof(NTFS_FILE_RECORD_OUTPUT_BUFFER) +
(sizeof(ntfsVolData.BytesPerFileRecordSegment)- 1), &dwWritten, NULL);

            // More data will make DeviceIoControl() fails...
            /*if(bDioControl == 0)
            {
                  wprintf(L"DeviceIoControl(...,FSCTL_GET_NTFS_FILE_RECORD,...)
failed!\n");
                  ErrorMessage(GetLastError());
                  exit(1);
            }*/

            // FSCTL_GET_NTFS_FILE_RECORD retrieves one MFT entry
            // FILE_RECORD_HEADER is the Base struct for the MFT entry
            // that we will work from
            PFILE_RECORD_HEADER p_file_record_header =
      (PFILE_RECORD_HEADER)output_buffer->FileRecordBuffer;
      }

      // Let verify
      wprintf(L"i\'s count = %u\n", i);

      //======================
      if(CloseHandle(hVolume) != 0)
            wprintf(L"hVolume handle was closed successfully!\n");
      else
      {
            wprintf(L"Failed to close hVolume handle!\n");
            ErrorMessage(GetLastError());
      }
      // De-allocate the memory by malloc()
      free(output_buffer);

      return 0;
}

void ErrorMessage(DWORD dwCode)
{
    // get the error code...
    DWORD dwErrCode = dwCode;
      DWORD dwNumChar;
```

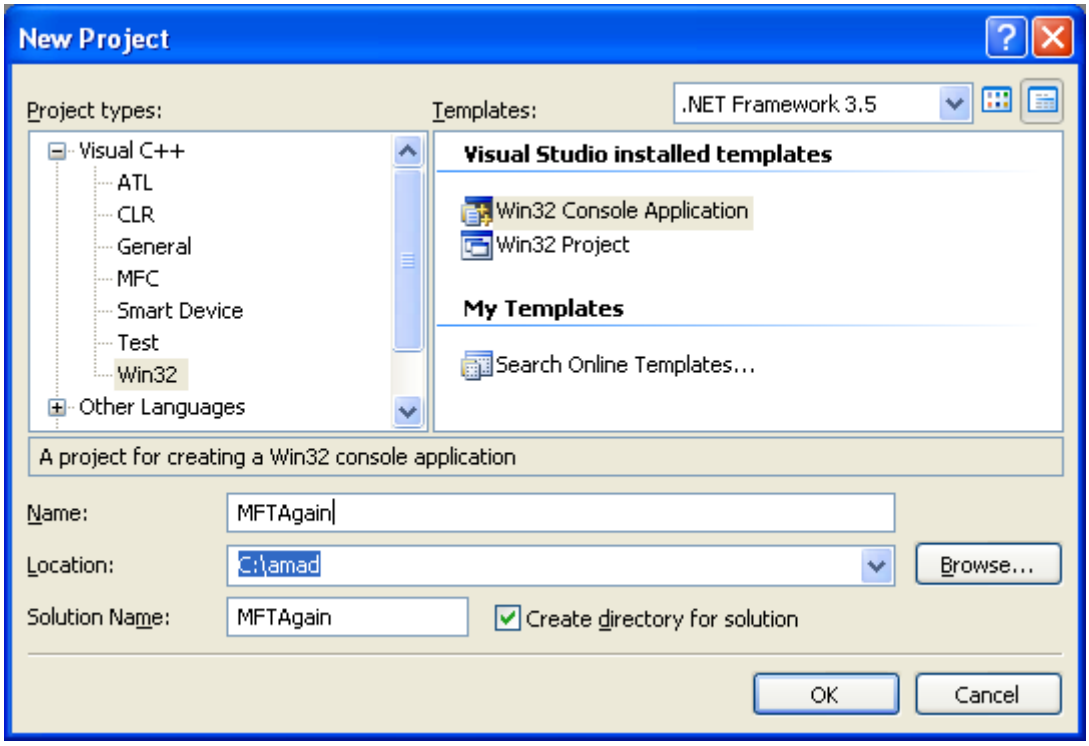
```
    LPWSTR szErrString = NULL;  // will be allocated and filled by FormatMessage

    dwNumChar = FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM, // use windows internal message
table
                0,         // 0 since source is internal message table
                dwErrCode, // this is the error code number
                0,         // auto-determine language to use
                (LPWSTR)&szErrString, // the messsage
                0,                    // min size for buffer
                0 );                  // since getting message from system
tables
      if(dwNumChar == 0)
            wprintf(L"FormatMessage() failed, error %u\n", GetLastError());
      //else
      //    wprintf(L"FormatMessage() should be fine!\n");

     wprintf(L"Error code %u:\n  %s\n", dwErrCode, szErrString) ;

      // This buffer used by FormatMessage()
    if(LocalFree(szErrString) != NULL)
            wprintf(L"Failed to free up the buffer, error %u\n",
GetLastError());
      //else
      //    wprintf(L"Buffer has been freed\n");
  }
```

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

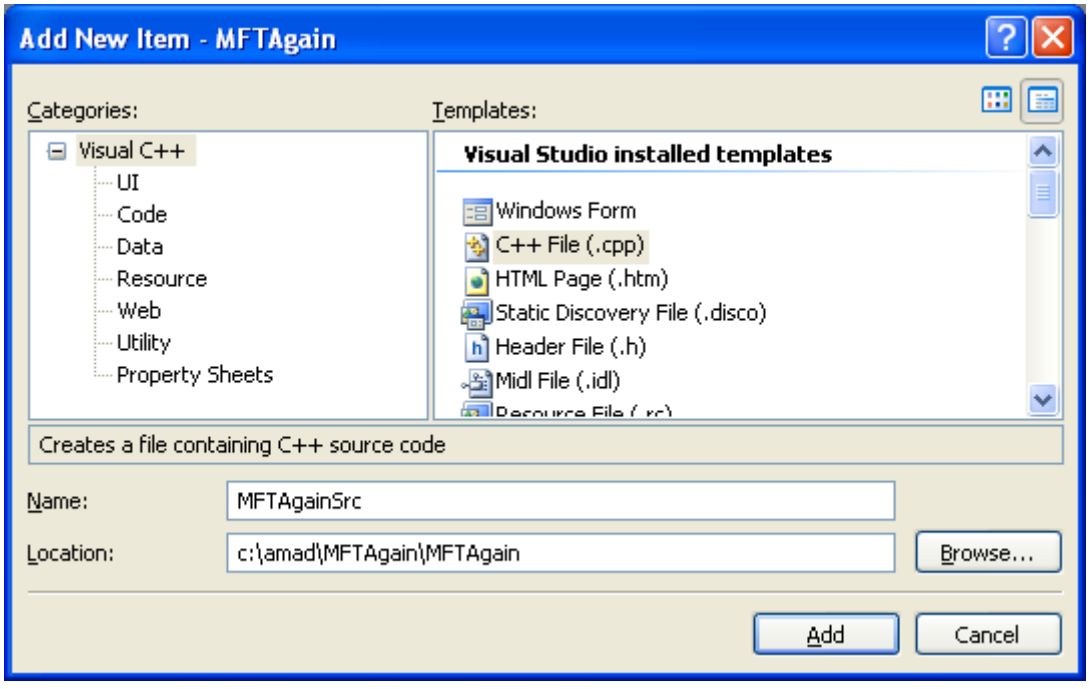


Listing the Deleted Files from Master File Table (MFT)

The following program example uses undocumented Windows types that can be found in the Internet domain and a complete version is available as an open source used by Linux/UNIX to read the Windows NTFS MFT.
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

```
// Not using winioctl.h lol!
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "ntfs.h"

// Global variables
ULONG BytesPerFileRecord;
HANDLE hVolume;
BOOT_BLOCK bootb;
PFILE_RECORD_HEADER MFT;

// Template for padding
template <class T1, class T2> inline T1* Padd(T1* p, T2 n)
{
	return (T1*)((char *)p + n);
}

ULONG RunLength(PUCHAR run)
{
	wprintf(L"In RunLength()...\n");
	return (*run & 0xf) + ((*run >> 4) & 0xf) + 1;
}

LONGLONG RunLCN(PUCHAR run)
{
	LONG i = 0;
	UCHAR n1 = 0 , n2 = 0;
	LONGLONG lcn = 0;

	wprintf(L"In RunLCN()...\n");
	n1 = *run & 0xf;
	n2 = (*run >> 4) & 0xf;

	lcn = n2 == 0 ? 0 : CHAR(run[n1 + n2]);

	for (i = n1 + n2 - 1; i > n1; i--)
		lcn = (lcn << 8) + run[i];
	return lcn;
}

ULONGLONG RunCount(PUCHAR run)
{
	UCHAR n = *run & 0xf;
	ULONGLONG count = 0;
	ULONG i;

	wprintf(L"In RunCount()...\n");

	for (i = n; i > 0; i--)
		count = (count << 8) + run[i];

	return count;
```

```
}

BOOL FindRun(PNONRESIDENT_ATTRIBUTE attr, ULONGLONG vcn, PULONGLONG lcn,
PULONGLONG count)
{
	PUCHAR run = NULL;
	*lcn = 0;
	ULONGLONG base = attr->LowVcn;

	wprintf(L"In FindRun()...\n");

	if (vcn < attr->LowVcn || vcn > attr->HighVcn)
		return FALSE;

	for(run = PUCHAR(Padd(attr, attr->RunArrayOffset));   *run != 0;  run +=
RunLength(run))
	{
		*lcn += RunLCN(run);
		*count = RunCount(run);

		if (base <= vcn && vcn < base + *count)
		{
			*lcn = RunLCN(run) == 0 ? 0 : *lcn + vcn - base;
			*count -= ULONG(vcn - base);
			return TRUE;
		}
		else
			base += *count;
	}
	return FALSE;
}

PATTRIBUTE FindAttribute(PFILE_RECORD_HEADER file,ATTRIBUTE_TYPE type, PWSTR
name)
{
	PATTRIBUTE attr = NULL;

	wprintf(L"FindAttribute() - Finding attributes...\n");

	for (attr = PATTRIBUTE(Padd(file, file->AttributesOffset));
		attr->AttributeType != -1;attr = Padd(attr, attr->Length))
	{
		if (attr->AttributeType == type)
		{
			if (name == 0 && attr->NameLength == 0)
				return attr;
			if (name != 0 && wcslen(name) == attr->NameLength &&
_wcsicmp(name,
				PWSTR(Padd(attr, attr->NameOffset))) == 0)
			return attr;
		}
	}
	return 0;
}

VOID FixupUpdateSequenceArray(PFILE_RECORD_HEADER file)
```

```
{
	ULONG i = 0;
	PUSHORT usa = PUSHORT(Padd(file, file->Ntfs.UsaOffset));
	PUSHORT sector = PUSHORT(file);

	wprintf(L"In FixupUpdateSequenceArray()...\n");
	for (i = 1; i < file->Ntfs.UsaCount; i++)
	{
		sector[255] = usa[i];
		sector += 256;
	}
}

VOID ReadSector(ULONGLONG sector, ULONG count, PVOID buffer)
{
	ULARGE_INTEGER offset;
	OVERLAPPED overlap = {0};
	ULONG n;

	wprintf(L"ReadSector() - Reading the sector...\n");
	wprintf(L"Sector: %lu\n", sector);

	offset.QuadPart = sector * bootb.BytesPerSector;
	overlap.Offset = offset.LowPart;
	overlap.OffsetHigh = offset.HighPart;
	ReadFile(hVolume, buffer, count * bootb.BytesPerSector, &n, &overlap);
}

VOID ReadLCN(ULONGLONG lcn, ULONG count, PVOID buffer)
{
	wprintf(L"\nReadLCN() - Reading the LCN, LCN: 0X%.8X\n", lcn);
	ReadSector(lcn * bootb.SectorsPerCluster,count * bootb.SectorsPerCluster,
buffer);
}

// Non resident attributes
VOID ReadExternalAttribute(PNONRESIDENT_ATTRIBUTE attr,ULONGLONG vcn, ULONG
count, PVOID buffer)
{
	ULONGLONG lcn, runcount;
	ULONG readcount, left;
	PUCHAR bytes = PUCHAR(buffer);

	wprintf(L"ReadExternalAttribute() - Reading the Non resident
attributes...\n");

	for(left = count; left > 0; left -= readcount)
	{
		FindRun(attr, vcn, &lcn, &runcount);
		readcount = ULONG(min(runcount, left));
		ULONG n = readcount * bootb.BytesPerSector *
bootb.SectorsPerCluster;

		if(lcn == 0)
			memset(bytes, 0, n);
		else
```

```
            {
                  ReadLCN(lcn, readcount, bytes);
                  wprintf(L"LCN: 0X%.8X\n", lcn);
            }
            vcn += readcount;
            bytes += n;
      }
}

ULONG AttributeLength(PATTRIBUTE attr)
{
      wprintf(L"In AttributeLength()...\n");
      return attr->Nonresident == FALSE ?
            PRESIDENT_ATTRIBUTE(attr)->ValueLength :
ULONG(PNONRESIDENT_ATTRIBUTE(attr)->DataSize);
}

ULONG AttributeLengthAllocated(PATTRIBUTE attr)
{
      wprintf(L"\nIn AttributeLengthAllocated()...\n");
      return attr->Nonresident == FALSE ?
            PRESIDENT_ATTRIBUTE(attr)->ValueLength :
ULONG(PNONRESIDENT_ATTRIBUTE(attr)->AllocatedSize);
}

VOID ReadAttribute(PATTRIBUTE attr, PVOID buffer)
{
      PRESIDENT_ATTRIBUTE rattr = NULL;
      PNONRESIDENT_ATTRIBUTE nattr = NULL;

      wprintf(L"ReadAttribute() - Reading the attributes...\n");
      if (attr->Nonresident == FALSE)
      {
            wprintf(L"Resident attribute...\n");
            rattr = PRESIDENT_ATTRIBUTE(attr);
            memcpy(buffer, Padd(rattr, rattr->ValueOffset), rattr->ValueLength);
      }
      else
      {
            wprintf(L"Non-resident attribute...\n");
            nattr = PNONRESIDENT_ATTRIBUTE(attr);
            ReadExternalAttribute(nattr, 0, ULONG(nattr->HighVcn) + 1, buffer);
      }
}

VOID ReadVCN(PFILE_RECORD_HEADER file, ATTRIBUTE_TYPE type,ULONGLONG vcn, ULONG
count, PVOID buffer)
{
      PATTRIBUTE attrlist = NULL;
      PNONRESIDENT_ATTRIBUTE attr = PNONRESIDENT_ATTRIBUTE(FindAttribute(file,
type, 0));

      wprintf(L"In ReadVCN()...\n");
      if (attr == 0 || (vcn < attr->LowVcn || vcn > attr->HighVcn))
      {
            // Support for huge files
```

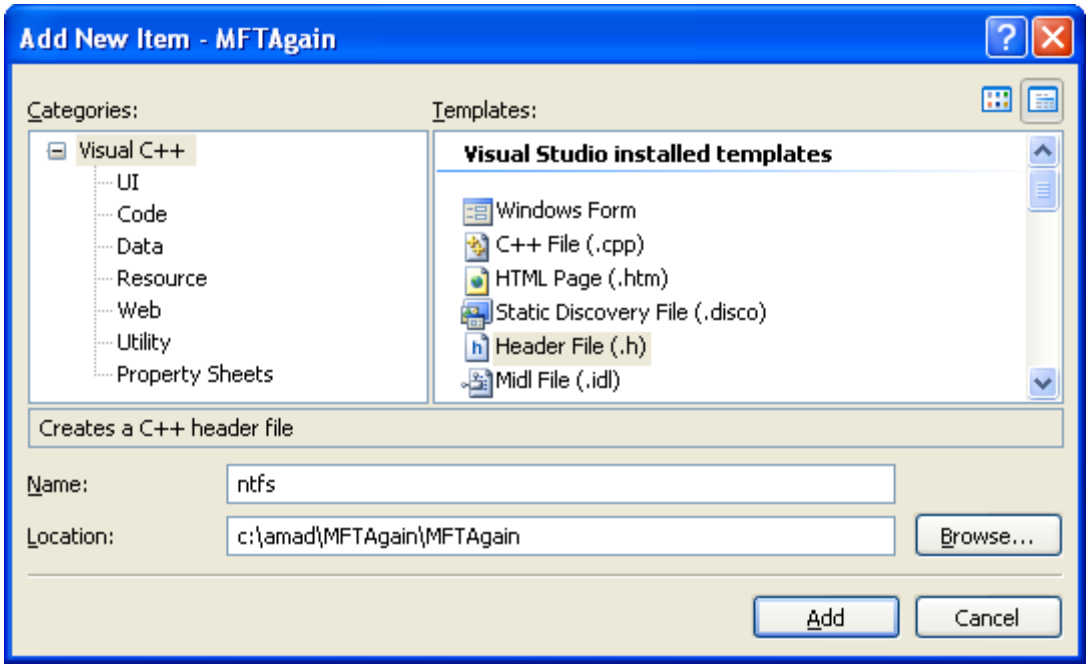
```
            attrlist = FindAttribute(file, AttributeAttributeList, 0);
            DebugBreak();
      }
      ReadExternalAttribute(attr, vcn, count, buffer);
}

VOID ReadFileRecord(ULONG index, PFILE_RECORD_HEADER file)
{
      ULONG clusters = bootb.ClustersPerFileRecord;

      wprintf(L"ReadFileRecord() - Reading the file records..\n");
      if (clusters > 0x80)
            clusters = 1;

      PUCHAR p = new UCHAR[bootb.BytesPerSector* bootb.SectorsPerCluster *
clusters];
      ULONGLONG vcn = ULONGLONG(index) *
BytesPerFileRecord/bootb.BytesPerSector/bootb.SectorsPerCluster;
      ReadVCN(MFT, AttributeData, vcn, clusters, p);
      LONG m = (bootb.SectorsPerCluster *
bootb.BytesPerSector/BytesPerFileRecord) - 1;
      ULONG n = m > 0 ? (index & m) : 0;
      memcpy(file, p + n * BytesPerFileRecord, BytesPerFileRecord);
      delete [] p;
      FixupUpdateSequenceArray(file);
}

VOID LoadMFT()
{
      wprintf(L"In LoadMFT() - Loading MFT...\n");

      BytesPerFileRecord = bootb.ClustersPerFileRecord < 0x80
            ? bootb.ClustersPerFileRecord* bootb.SectorsPerCluster
            * bootb.BytesPerSector: 1 << (0x100 - bootb.ClustersPerFileRecord);

      wprintf(L"\nBytes Per File Record = %u\n\n", BytesPerFileRecord);
      wprintf(L"======THESE INFO ARE NOT ACCURATE FOR DISPLAY LOL!=====\n");
      wprintf(L"bootb.BootSectors = %u\n", bootb.BootSectors);
      wprintf(L"bootb.BootSignature = %u\n", bootb.BootSignature);
      wprintf(L"bootb.BytesPerSector = %u\n", bootb.BytesPerSector);
      wprintf(L"bootb.ClustersPerFileRecord = %u\n",
bootb.ClustersPerFileRecord);
      wprintf(L"bootb.ClustersPerIndexBlock = %u\n",
bootb.ClustersPerIndexBlock);
      wprintf(L"bootb.Code = %u\n", bootb.Code);
      wprintf(L"bootb.Format = %u\n", bootb.Format);
      wprintf(L"bootb.Jump = %u\n", bootb.Jump);
      wprintf(L"bootb.Mbz1 = %u\n", bootb.Mbz1);
      wprintf(L"bootb.Mbz2 = %u\n", bootb.Mbz2);
      wprintf(L"bootb.Mbz3 = %u\n", bootb.Mbz3);
      wprintf(L"bootb.MediaType = 0X%X\n", bootb.MediaType);
      wprintf(L"bootb.Mft2StartLcn = 0X%.8X\n", bootb.Mft2StartLcn);
      wprintf(L"bootb.MftStartLcn = 0X%.8X\n", bootb.MftStartLcn);
      wprintf(L"bootb.NumberOfHeads = %u\n", bootb.NumberOfHeads);
      wprintf(L"bootb.PartitionOffset = %lu\n", bootb.PartitionOffset);
      wprintf(L"bootb.SectorsPerCluster = %u\n", bootb.SectorsPerCluster);
```

```
	wprintf(L"bootb.SectorsPerTrack = %u\n", bootb.SectorsPerTrack);
	wprintf(L"bootb.TotalSectors = %lu\n", bootb.TotalSectors);
	wprintf(L"bootb.VolumeSerialNumber = 0X%.8X%.8X\n\n",
bootb.VolumeSerialNumber.HighPart, bootb.VolumeSerialNumber.HighPart);


	MFT = PFILE_RECORD_HEADER(new UCHAR[BytesPerFileRecord]);

	ReadSector((bootb.MftStartLcn)*(bootb.SectorsPerCluster),
(BytesPerFileRecord)/(bootb.BytesPerSector), MFT);
	FixupUpdateSequenceArray(MFT);
}

BOOL bitset(PUCHAR bitmap, ULONG i)
{
	return (bitmap[i >> 3] & (1 << (i & 7))) != 0;
}

VOID FindDeleted()
{
	PATTRIBUTE attr = FindAttribute(MFT, AttributeBitmap, 0);
	PUCHAR bitmap = new UCHAR[AttributeLengthAllocated(attr)];
	ReadAttribute(attr, bitmap);
	ULONG n = AttributeLength(FindAttribute(MFT, AttributeData,
0))/BytesPerFileRecord;

	wprintf(L"FindDeleted() - Finding the deleted files...\n");

	PFILE_RECORD_HEADER file = PFILE_RECORD_HEADER(new
UCHAR[BytesPerFileRecord]);

	for(ULONG i = 0; i < n; i++)
	{
		if (bitset(bitmap, i))
			continue;

		ReadFileRecord(i, file);

		if (file->Ntfs.Type == 'ELIF' && (file->Flags & 1) == 0)
		{
			attr = FindAttribute(file, AttributeFileName, 0);
			if (attr == 0)
				continue;

			PFILENAME_ATTRIBUTE name =
PFILENAME_ATTRIBUTE(Padd(attr,PRESIDENT_ATTRIBUTE(attr)->ValueOffset));

			// * means the width/precision was supplied in the argument
list
			// ws ~ wide character string
			wprintf(L"\n%10u %u %.*s\n\n", i, int(name->NameLength),
int(name->NameLength), name->Name);
			// To see the very long output short, uncomment the following
line
			// _getwch();
		}
```

```
	}
}

VOID DumpData(ULONG index, WCHAR* filename)
{
	PATTRIBUTE attr = NULL;
	HANDLE hFile = NULL;
	PFILE_RECORD_HEADER file = PFILE_RECORD_HEADER(new
UCHAR[BytesPerFileRecord]);
	ULONG n;


	ReadFileRecord(index, file);

	wprintf(L"Dumping the data...\n");

	if (file->Ntfs.Type != 'ELIF')
		return;

	attr = FindAttribute(file, AttributeData, 0);
	if (attr == 0)
		return;

	PUCHAR buf = new UCHAR[AttributeLengthAllocated(attr)];
	ReadAttribute(attr, buf);

	hFile = CreateFile((LPCWSTR)filename, GENERIC_WRITE, 0, 0,CREATE_ALWAYS,
0, 0);

	if(hFile == INVALID_HANDLE_VALUE)
	{
		wprintf(L"CreateFile() failed, error %u\n", GetLastError());
		return;
	}

	if(WriteFile(hFile, buf, AttributeLength(attr), &n, 0) == 0)
	{
		wprintf(L"WriteFile() failed, error %u\n", GetLastError());
		return;
	}

	CloseHandle(hFile);
	delete [] buf;
}

int wmain(int argc, WCHAR **argv)
{
	// Default primary partition
	WCHAR drive[] = L"\\\\.\\C:";
	ULONG n;

	// No argument supplied
	if (argc < 2)
	{
		wprintf(L"Usage:\n");
		wprintf(L"Find deleted files: %s <primary_partition>\n", argv[0]);
```

```
            wprintf(L"Read the file records: %s <primary_partition> <index>
<file_name>\n", argv[0]);
            // Just exit
            exit(1);
      }
      // More code to stop the user from entering the non-primary partition

      // Read the user input
      drive[4] = *argv[1];

      // Get the handle to the primary partition/volume/physical disk
      hVolume = CreateFile(
            drive,
            GENERIC_READ,
            FILE_SHARE_READ | FILE_SHARE_WRITE,
            0,
            OPEN_EXISTING,
            0,
            0);

      if(hVolume == INVALID_HANDLE_VALUE)
      {
            wprintf(L"CreateFile() failed, error %u\n", GetLastError());
            exit(1);
      }

      // Reads data from the specified input/output (I/O) device -
volume/physical disk
      if(ReadFile(hVolume, &bootb, sizeof bootb, &n, 0) == 0)
      {
            wprintf(L"ReadFile() failed, error %u\n", GetLastError());
            exit(1);
      }

      LoadMFT();

      // The primary partition supplied else
      // default C:\ will be used
      if (argc == 2)
            FindDeleted();

      // Need to convert the recovered filename to long file name
      // Not implemented here. It is 8.3 file name format

      // The primary partition, index and file name to be recovered
      // are supplied
      if (argc == 4)
            DumpData(wcstoul(argv[2], 0, 0), argv[3]);

      CloseHandle(hVolume);
      return 0;
}
```

Add the ntfs.h header file.



Add the following source code.

```
///ntfs.h
// These types are not available in MSDN documentation
// It is taken from Internet and Linux documentation
// and not the whole code...
// Copyrights and trademarks must go to the original
// authors and/or publishers
typedef struct {
	ULONG Type;
	USHORT UsaOffset;
	USHORT UsaCount;
	USN Usn;
} NTFS_RECORD_HEADER, *PNTFS_RECORD_HEADER;

typedef struct {
	NTFS_RECORD_HEADER Ntfs;
	USHORT SequenceNumber;
	USHORT LinkCount;
	USHORT AttributesOffset;
	USHORT Flags; // 0x0001 = InUse, 0x0002= Directory
	ULONG BytesInUse;
	ULONG BytesAllocated;
	ULONGLONG BaseFileRecord;
	USHORT NextAttributeNumber;
} FILE_RECORD_HEADER, *PFILE_RECORD_HEADER;

typedef enum {
	AttributeStandardInformation = 0x10,
	AttributeAttributeList = 0x20,
	AttributeFileName = 0x30,
	AttributeObjectId = 0x40,
	AttributeSecurityDescriptor = 0x50,
```

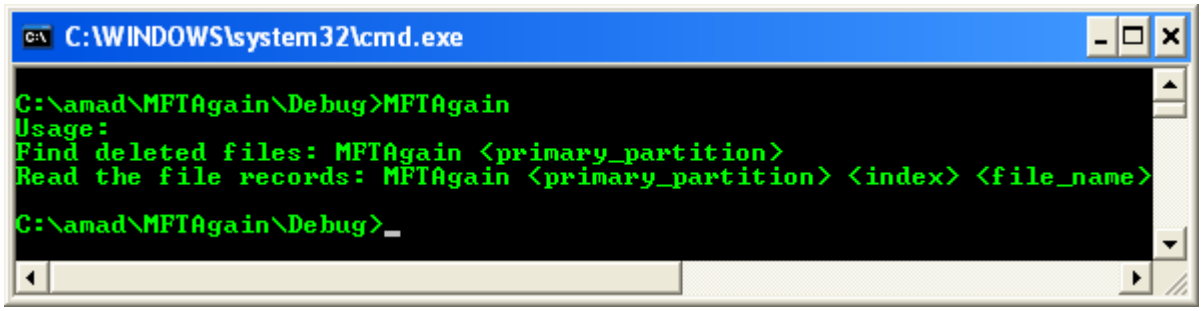
```
        AttributeVolumeName = 0x60,
        AttributeVolumeInformation = 0x70,
        AttributeData = 0x80,
        AttributeIndexRoot = 0x90,
        AttributeIndexAllocation = 0xA0,
        AttributeBitmap = 0xB0,
        AttributeReparsePoint = 0xC0,
        AttributeEAInformation = 0xD0,
        AttributeEA = 0xE0,
        AttributePropertySet = 0xF0,
        AttributeLoggedUtilityStream = 0x100
} ATTRIBUTE_TYPE, *PATTRIBUTE_TYPE;

typedef struct {
        ATTRIBUTE_TYPE AttributeType;
        ULONG Length;
        BOOLEAN Nonresident;
        UCHAR NameLength;
        USHORT NameOffset;
        USHORT Flags; // 0x0001 = Compressed
        USHORT AttributeNumber;
} ATTRIBUTE, *PATTRIBUTE;

typedef struct {
        ATTRIBUTE Attribute;
        ULONG ValueLength;
        USHORT ValueOffset;
        USHORT Flags; // 0x0001 = Indexed
} RESIDENT_ATTRIBUTE, *PRESIDENT_ATTRIBUTE;

typedef struct {
        ATTRIBUTE Attribute;
        ULONGLONG LowVcn;
        ULONGLONG HighVcn;
        USHORT RunArrayOffset;
        UCHAR CompressionUnit;
        UCHAR AlignmentOrReserved[5];
        ULONGLONG AllocatedSize;
        ULONGLONG DataSize;
        ULONGLONG InitializedSize;
        ULONGLONG CompressedSize; // Only when compressed
} NONRESIDENT_ATTRIBUTE, *PNONRESIDENT_ATTRIBUTE;

typedef struct {
        ULONGLONG CreationTime;
        ULONGLONG ChangeTime;
        ULONGLONG LastWriteTime;
        ULONGLONG LastAccessTime;
        ULONG FileAttributes;
        ULONG AlignmentOrReservedOrUnknown[3];
        ULONG QuotaId;                  // NTFS 3.0
        ULONG SecurityId;       // NTFS 3.0
        ULONGLONG QuotaCharge;  // NTFS 3.0
        USN Usn;                        // NTFS 3.0
} STANDARD_INFORMATION, *PSTANDARD_INFORMATION;
```

```
typedef struct {
	ATTRIBUTE_TYPE AttributeType;
	USHORT Length;
	UCHAR NameLength;
	UCHAR NameOffset;
	ULONGLONG LowVcn;
	ULONGLONG FileReferenceNumber;
	USHORT AttributeNumber;
	USHORT AlignmentOrReserved[3];
} ATTRIBUTE_LIST, *PATTRIBUTE_LIST;

typedef struct {
	ULONGLONG DirectoryFileReferenceNumber;		//
	ULONGLONG CreationTime;		// Saved when filename last changed
	ULONGLONG ChangeTime;		//
	ULONGLONG LastWriteTime;	//
	ULONGLONG LastAccessTime;	//
	ULONGLONG AllocatedSize;	//
	ULONGLONG DataSize;			//
	ULONG FileAttributes;		//
	ULONG AlignmentOrReserved;	//
	UCHAR NameLength;			//
	UCHAR NameType;				// 0x01 = Long, 0x02 = Short
	WCHAR Name[1];				//
} FILENAME_ATTRIBUTE, *PFILENAME_ATTRIBUTE;

typedef struct {
	GUID ObjectId;
	union {
		struct {
			GUID BirthVolumeId;
			GUID BirthObjectId;
			GUID DomainId;
		};
		UCHAR ExtendedInfo[48];
	};
} OBJECTID_ATTRIBUTE, *POBJECTID_ATTRIBUTE;

typedef struct {
	ULONG Unknown[2];
	UCHAR MajorVersion;
	UCHAR MinorVersion;
	USHORT Flags;
} VOLUME_INFORMATION, *PVOLUME_INFORMATION;

typedef struct {
	ULONG EntriesOffset;
	ULONG IndexBlockLength;
	ULONG AllocatedSize;
	ULONG Flags; // 0x00 = Small directory, 0x01 = Large directory
} DIRECTORY_INDEX, *PDIRECTORY_INDEX;

typedef struct {
	ULONGLONG FileReferenceNumber;
	USHORT Length;
	USHORT AttributeLength;
```

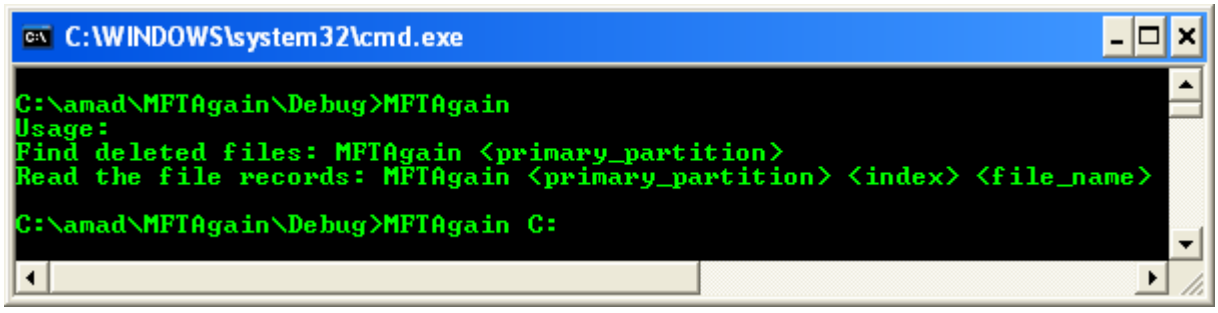
```
	ULONG Flags; // 0x01 = Has trailing VCN, 0x02 = Last entry
	// FILENAME_ATTRIBUTE Name;
	// ULONGLONG Vcn; // VCN in IndexAllocation of earlier entries
} DIRECTORY_ENTRY, *PDIRECTORY_ENTRY;

typedef struct {
	ATTRIBUTE_TYPE Type;
	ULONG CollationRule;
	ULONG BytesPerIndexBlock;
	ULONG ClustersPerIndexBlock;
	DIRECTORY_INDEX DirectoryIndex;
} INDEX_ROOT, *PINDEX_ROOT;

typedef struct {
	NTFS_RECORD_HEADER Ntfs;
	ULONGLONG IndexBlockVcn;
	DIRECTORY_INDEX DirectoryIndex;
} INDEX_BLOCK_HEADER, *PINDEX_BLOCK_HEADER;

typedef struct {
	ULONG ReparseTag;
	USHORT ReparseDataLength;
	USHORT Reserved;
	UCHAR ReparseData[1];
} REPARSE_POINT, *PREPARSE_POINT;

typedef struct {
	ULONG EaLength;
	ULONG EaQueryLength;
} EA_INFORMATION, *PEA_INFORMATION;

typedef struct {
	ULONG NextEntryOffset;
	UCHAR Flags;
	UCHAR EaNameLength;
	USHORT EaValueLength;
	CHAR EaName[1];
	// UCHAR EaData[];
} EA_ATTRIBUTE, *PEA_ATTRIBUTE;

typedef struct {
	WCHAR AttributeName[64];
	ULONG AttributeNumber;
	ULONG Unknown[2];
	ULONG Flags;
	ULONGLONG MinimumSize;
	ULONGLONG MaximumSize;
} ATTRIBUTE_DEFINITION, *PATTRIBUTE_DEFINITION;

#pragma pack(push, 1)

typedef struct {
	UCHAR Jump[3];
	UCHAR Format[8];
	USHORT BytesPerSector;
	UCHAR SectorsPerCluster;
```

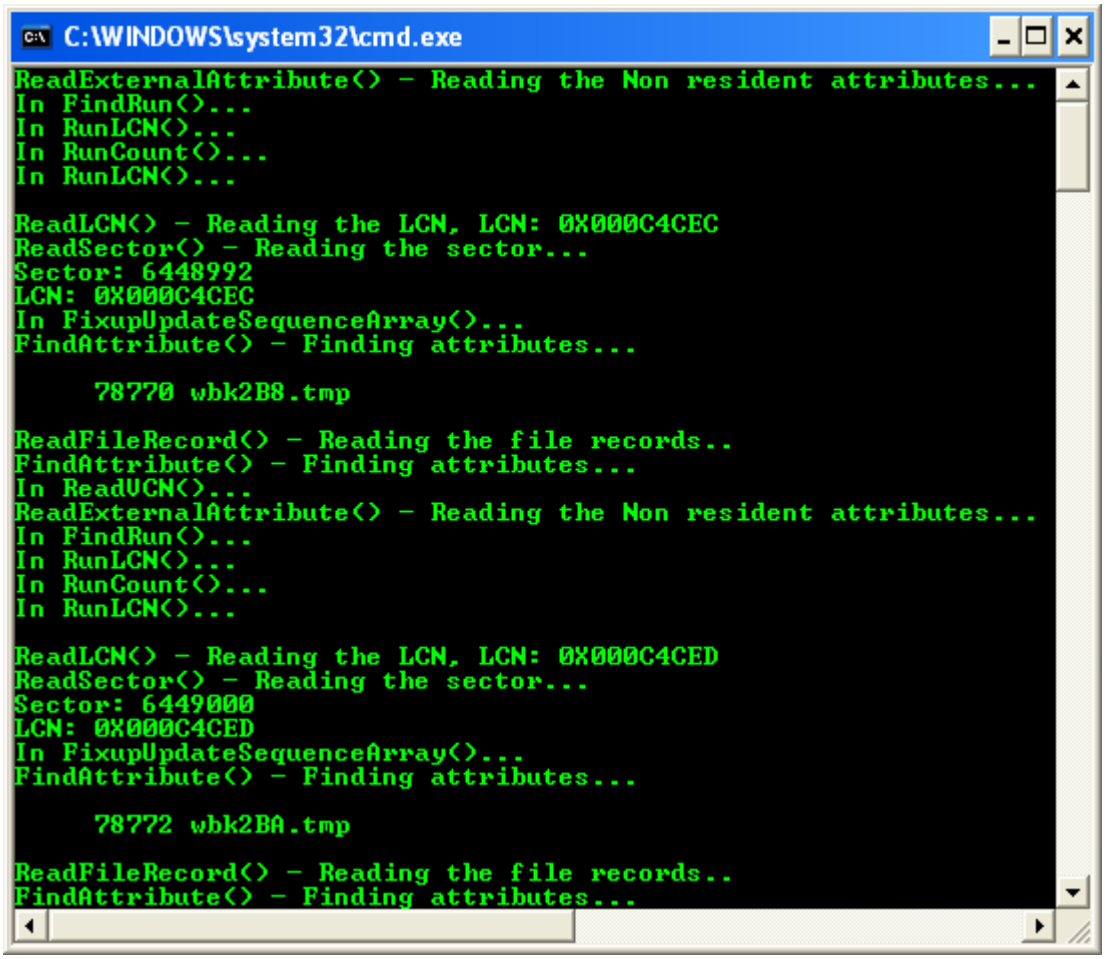
```
        USHORT BootSectors;
        UCHAR Mbz1;
        USHORT Mbz2;
        USHORT Reserved1;
        UCHAR MediaType;
        USHORT Mbz3;
        USHORT SectorsPerTrack;
        USHORT NumberOfHeads;
        ULONG PartitionOffset;
        ULONG Reserved2[2];
        ULONGLONG TotalSectors;
        ULONGLONG MftStartLcn;
        ULONGLONG Mft2StartLcn;
        ULONG ClustersPerFileRecord;
        ULONG ClustersPerIndexBlock;
        LARGE_INTEGER VolumeSerialNumber;
        UCHAR Code[0x1AE];
        USHORT BootSignature;

} BOOT_BLOCK, *PBOOT_BLOCK;

#pragma pack(pop)
```

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

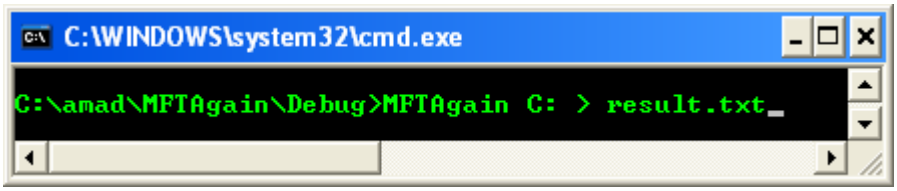


The following sample output run with the primary partition supplied as an argument.

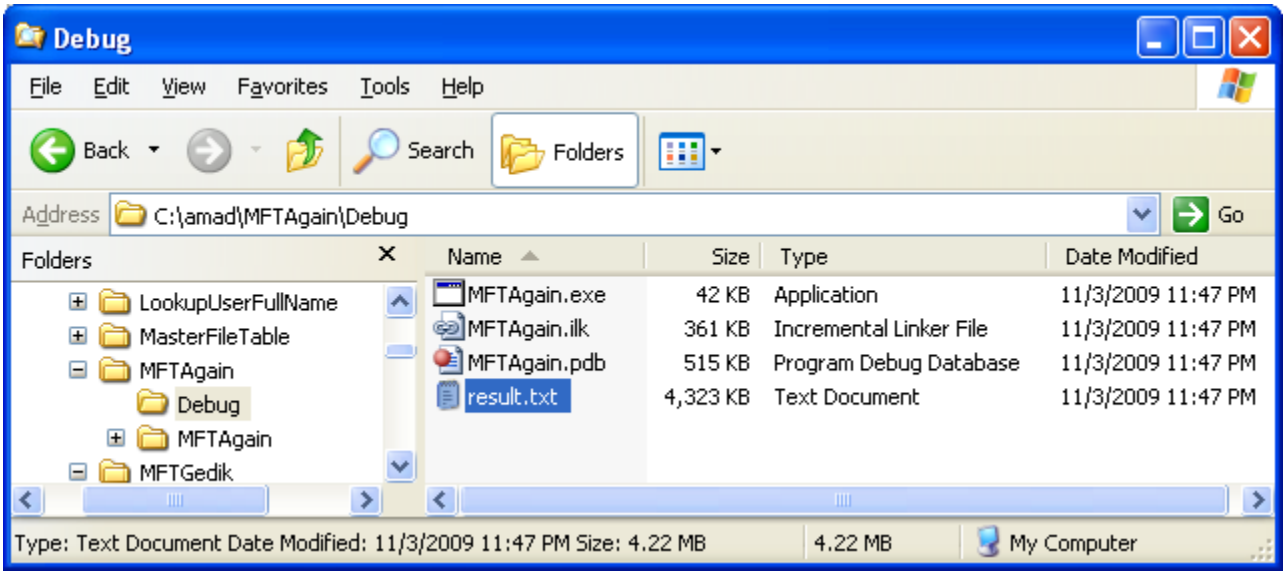


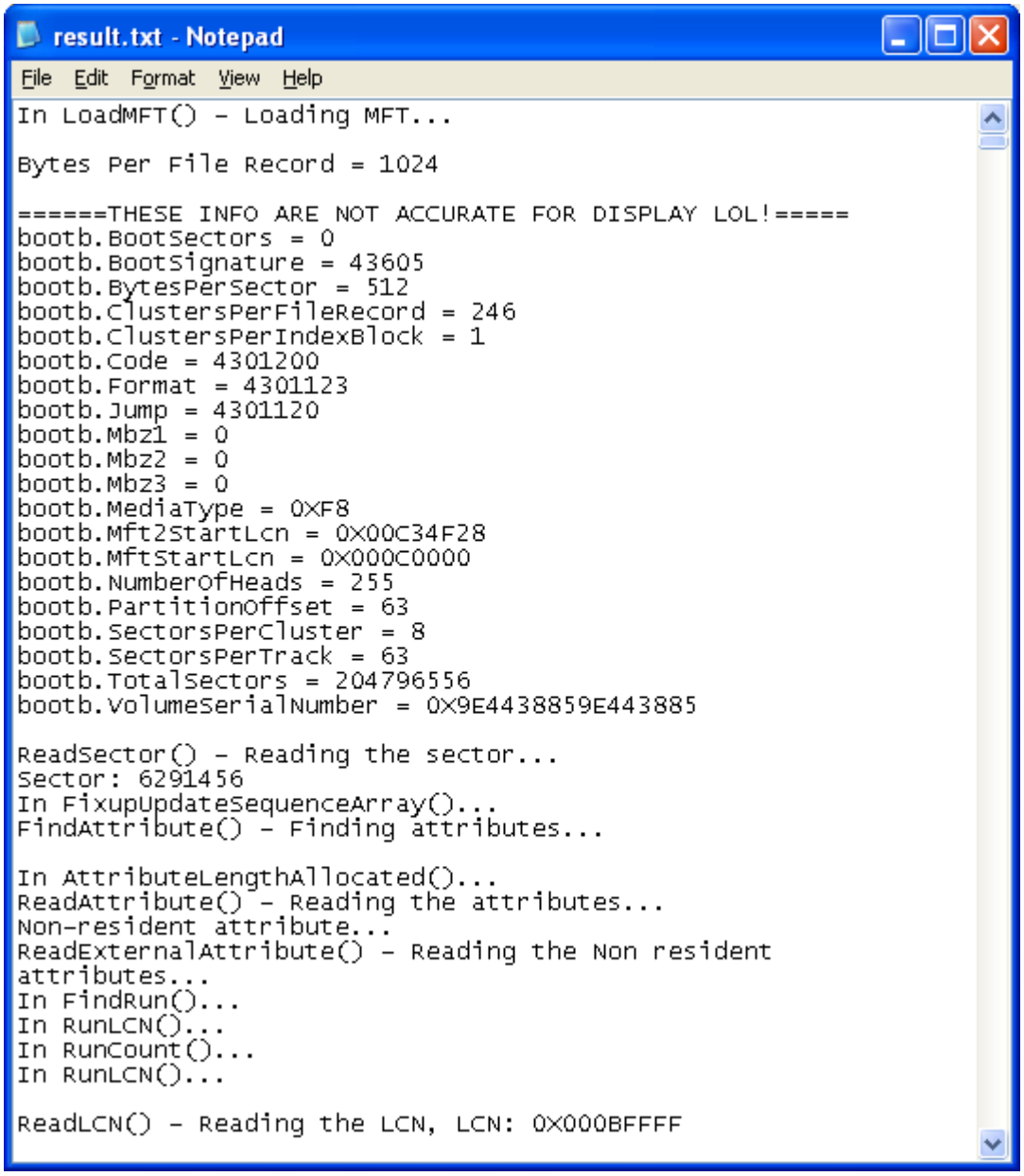


To save all the deleted file names you may want to redirect the output into a text file as shown below.

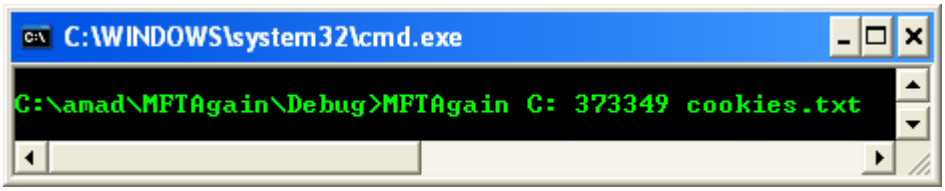


Then open the text file using any unformatted text editor such as WordPad.

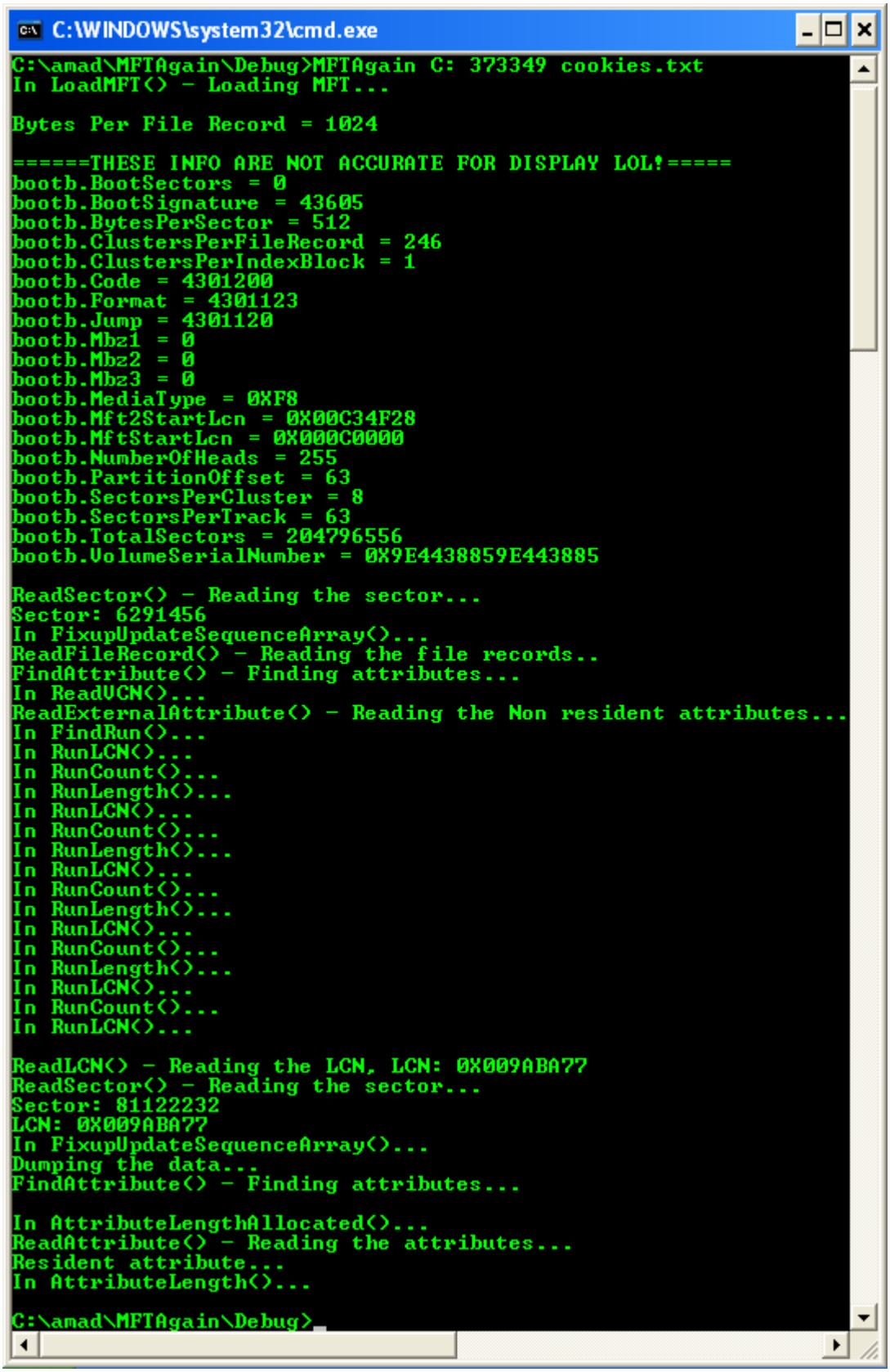

Debug
File Edit View Favorites Tools Help
Back Search Folders
Address C:\amad\MFTAgain\Debug
Go
Folders
LookupUserFullName
MasterFileTable
MFTAgain
Debug
MFTAgain
MFTGedik
Name Size Type Date Modified
MFTAgain.exe 42 KB Application 11/3/2009 11:47 PM
MFTAgain.ilk 361 KB Incremental Linker File 11/3/2009 11:47 PM
MFTAgain.pdb 515 KB Program Debug Database 11/3/2009 11:47 PM
result.txt 4,323 KB Text Document 11/3/2009 11:47 PM
Type: Text Document Date Modified: 11/3/2009 11:47 PM Size: 4.22 MB
4.22 MB
My Computer



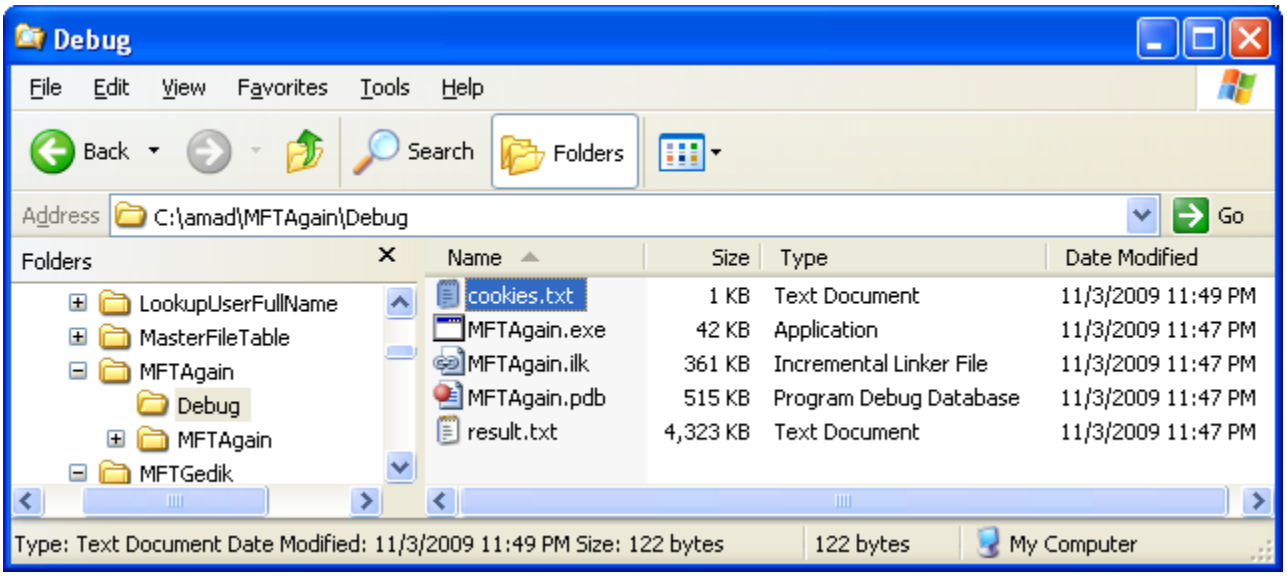
Next we will try to recover a file. We got the file name and index from the previous output.

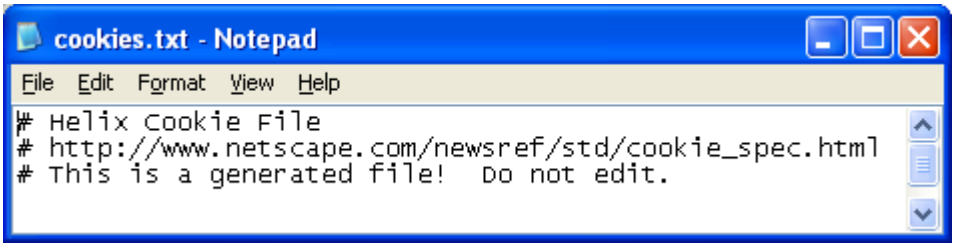


The following is a sample output.


C:\WINDOWS\system32\cmd.exe
C:\amad\MFTAgain\Debug>MFTAgain C: 373349 cookies.txt
In LoadMFT() - Loading MFT...
Bytes Per File Record = 1024
======THESE INFO ARE NOT ACCURATE FOR DISPLAY LOL!=====
bootb.BootSectors = 0
bootb.BootSignature = 43605
bootb.BytesPerSector = 512
bootb.ClustersPerFileRecord = 246
bootb.ClustersPerIndexBlock = 1
bootb.Code = 4301200
bootb.Format = 4301123
bootb.Jump = 4301120
bootb.Mbz1 = 0
bootb.Mbz2 = 0
bootb.Mbz3 = 0
bootb.MediaType = 0XF8
bootb.Mft2StartLcn = 0X00C34F28
bootb.MftStartLcn = 0X000C0000
bootb.NumberOfHeads = 255
bootb.PartitionOffset = 63
bootb.SectorsPerCluster = 8
bootb.SectorsPerTrack = 63
bootb.TotalSectors = 204796556
bootb.VolumeSerialNumber = 0X9E4438859E443885
ReadSector() - Reading the sector...
Sector: 6291456
In FixupUpdateSequenceArray()...
ReadFileRecord() - Reading the file records..
FindAttribute() - Finding attributes...
In ReadVCN()...
ReadExternalAttribute() - Reading the Non resident attributes...
In FindRun()...
In RunLCN()...
In RunCount()...
In RunLength()...
In RunLCN()...
In RunCount()...
In RunLength()...
In RunLCN()...
In RunCount()...
In RunLength()...
In RunLCN()...
In RunCount()...
In RunLength()...
In RunLCN()...
In RunCount()...
In RunLCN()...
ReadLCN() - Reading the LCN, LCN: 0X009ABA77
ReadSector() - Reading the sector...
Sector: 81122232
LCN: 0X009ABA77
In FixupUpdateSequenceArray()...
Dumping the data...
FindAttribute() - Finding attributes...
In AttributeLengthAllocated()...
ReadAttribute() - Reading the attributes...
Resident attribute...
In AttributeLength()...
C:\amad\MFTAgain\Debug>

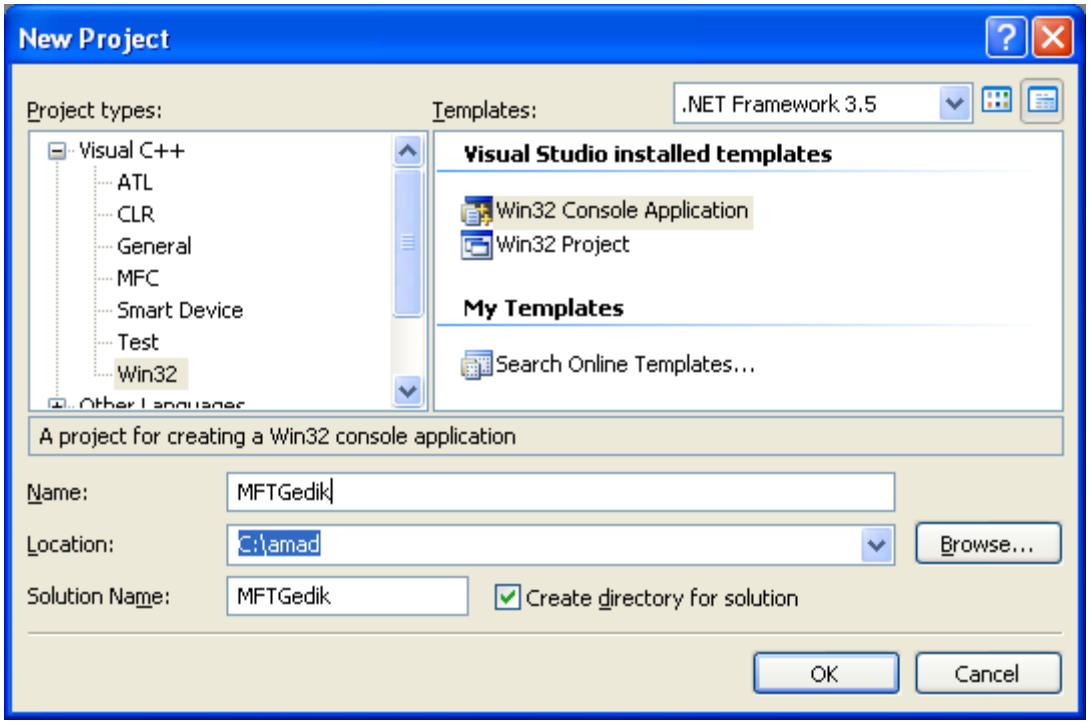
Next, the recovered file should be stored in the project's Debug folder.



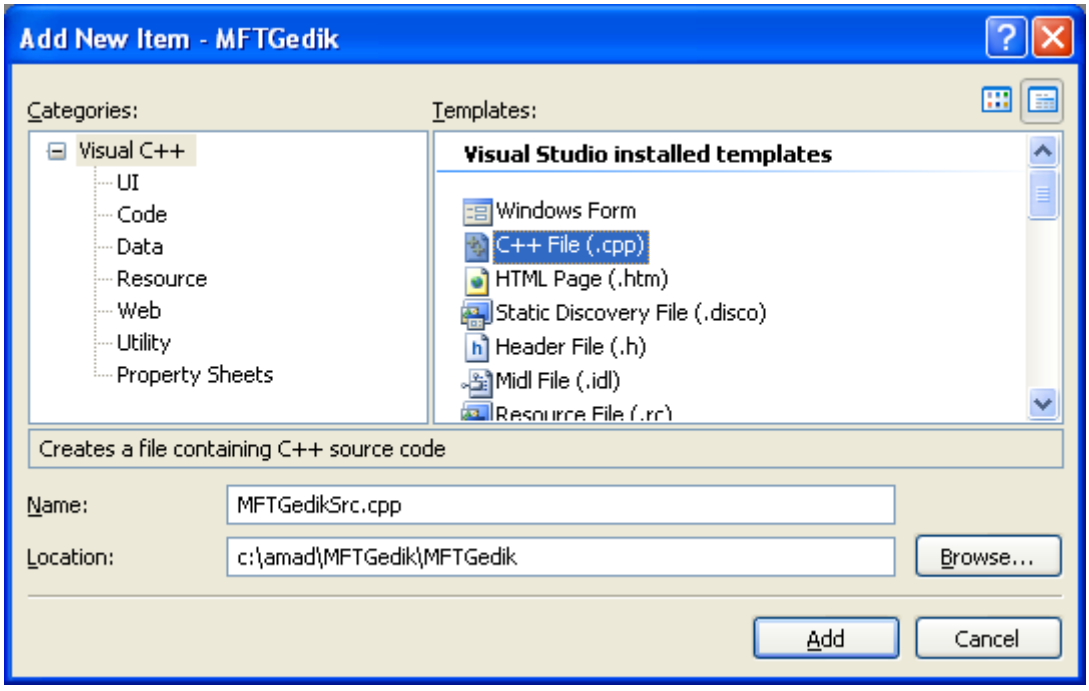


Another Day, Another MFT Program Example: List, Recover and Delete the Deleted Files from Master File Table

The following program example tries to extend the previous example by adding a 'feature' that can 'delete' a file in the Master File Table.
Create a new Win32 console application project and give a suitable project name.



Add the source file and give a suitable name.



Add the following source code.

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

```
#include "ntfs.h"

// Fixing the offset
#define FIXOFFSET(x, y) ((CHAR*)(x) + (y))

// Global variables. Not a good practice
ULONG     BytesPerFileRecord;
UINT     BytesPerCluster;
BOOT_BLOCK    BootBlk;
PFILE_RECORD_HEADER MFT;
HANDLE     hVolume;
FILE     *pFLog;

BOOL BitSet(PUCHAR Bitmap, ULONG Idx)
{
      return (Bitmap[Idx >> 3] & (1 << (Idx & 7))) != 0;
}

void FixupUpdateSequenceArray(PFILE_RECORD_HEADER FileRecord)
{
      PUSHORT UsAry = PUSHORT(FIXOFFSET(FileRecord, FileRecord-
>Ntfs.UsaOffset));
      PUSHORT Sector = PUSHORT(FileRecord);

      for (ULONG Idx = 1; Idx < FileRecord->Ntfs.UsaCount; Idx++)
      {
            // If( UsAry[0] != Sector[255] ) then this sector is corrupt or
broken
            Sector[255] = UsAry[Idx];
            Sector += 256;
      }
}

void ZeroSequenceArray(PFILE_RECORD_HEADER FileRecord)
{
      PUSHORT UsAry = PUSHORT(FIXOFFSET(FileRecord, FileRecord-
>Ntfs.UsaOffset));

      for (ULONG Idx = 1; Idx < FileRecord->Ntfs.UsaCount; Idx++)
      {
            UsAry[Idx] = 0x3030;
      }
}

void ReadSector(ULONGLONG Sector, ULONG Cnt, PVOID Buffer)
{
      ULARGE_INTEGER Offset;
      OVERLAPPED Overlap = {0};
      ULONG ReadBytes, CntIdx = 0, NeedReadByte = Cnt * BootBlk.BytesPerSector;

      // Assign the physical position.
      Offset.QuadPart   = Sector * BootBlk.BytesPerSector;
      // Set position to Overlap.
      Overlap.Offset   = Offset.LowPart;
      Overlap.OffsetHigh = Offset.HighPart;
```

```
	ReadFile(hVolume, Buffer, NeedReadByte, &ReadBytes, &Overlap);

	if(ReadBytes != NeedReadByte)
	{
		while(CntIdx < Cnt)
		{
			//Set position to Overlap.
			Overlap.Offset   = Offset.LowPart;
			Overlap.OffsetHigh = Offset.HighPart;

			ReadFile(hVolume, Buffer, BootBlk.BytesPerSector, &ReadBytes,
&Overlap);

			if(ReadBytes != BootBlk.BytesPerSector)
			{
				wprintf(L"Read Sector failed: %d:%d:%d\n",
					Offset.LowPart, Cnt * BootBlk.BytesPerSector,
ReadBytes);
				return;
			}

			Buffer = (UCHAR*)Buffer + BootBlk.BytesPerSector;
			// Update the physical position.
			Offset.QuadPart += BootBlk.BytesPerSector;
			++CntIdx;
		}
	}
	return;
}

void ReadLCN(ULONGLONG LCN, ULONG Cnt, PVOID Buffer)
{
	ReadSector(LCN * BootBlk.SectorsPerCluster, Cnt *
BootBlk.SectorsPerCluster, Buffer);
}

void WriteSector(ULONGLONG Sector, ULONG Cnt, PVOID Buffer)
{
	ULARGE_INTEGER Offset;
	OVERLAPPED Overlap = {0};
	ULONG Written;

	// Assign the physical position.
	Offset.QuadPart   = Sector * BootBlk.BytesPerSector;
	// Set position to Overlap.
	Overlap.Offset   = Offset.LowPart;
	Overlap.OffsetHigh = Offset.HighPart;
	WriteFile(hVolume, Buffer, Cnt * BootBlk.BytesPerSector, &Written,
&Overlap);

	if(Written != Cnt * BootBlk.BytesPerSector)
		wprintf(L"Wrote failed: %d:%d:%d\n", Overlap.Offset, Cnt *
BootBlk.BytesPerSector, Written);
}

void WriteLCN(ULONGLONG LCN, ULONG Cnt, PVOID Buffer)
```

```
{
	WriteSector(LCN * BootBlk.SectorsPerCluster, Cnt *
BootBlk.SectorsPerCluster, Buffer);
}

void ZeroLCN(ULONGLONG LCN, ULONG Cnt)
{
	ULONGLONG ZeroSectorNum=512;
	BYTE *p512Sector = new BYTE[(UINT)BootBlk.BytesPerSector *
(UINT)ZeroSectorNum];
	ULONGLONG SectorNum = Cnt * (UINT)BootBlk.SectorsPerCluster;
	ULONGLONG SectorSrtIdx = LCN * BootBlk.SectorsPerCluster, Idx;
	ULONGLONG SectorEndIdx = SectorSrtIdx + SectorNum;

	wprintf(L"->Sector Start Index: %d, Zero Sector number is %d\n",
(UINT)SectorSrtIdx, (UINT)SectorNum);

	memset(p512Sector, '0', (UINT)BootBlk.BytesPerSector *
(UINT)ZeroSectorNum);

	if(p512Sector)
	{
		Idx = SectorSrtIdx;

		while((SectorNum > 0) && (ZeroSectorNum > 0))
		{
			wprintf(L"\n%d--->", ZeroSectorNum);
			for(; (Idx < SectorEndIdx) && (SectorNum >= ZeroSectorNum);
Idx += ZeroSectorNum)
			{
				WriteSector(Idx, (UINT)ZeroSectorNum, p512Sector);
				SectorNum -= ZeroSectorNum;
				wprintf(L".");
			}
			ZeroSectorNum >>= 1;
		}

		if(SectorNum)
		{
			// Evaluates an expression and, when the result is false,
			// prints a diagnostic message and aborts the program.
			assert(SectorNum == 1);
			WriteSector(Idx, 1, p512Sector);
			wprintf(L"\n%3d--->.", 1);
		}

		wprintf(L"\n");
		delete [] p512Sector;
	}
}

ULONG AttributeLength(PATTRIBUTE Attr)
{
	return Attr->Nonresident == FALSE
		? PRESIDENT_ATTRIBUTE(Attr)->ValueLength:
ULONG(PNONRESIDENT_ATTRIBUTE(Attr)->DataSize);
```

```
}

ULONG AttributeLengthAllocated(PATTRIBUTE Attr)
{

	return Attr->Nonresident == FALSE? PRESIDENT_ATTRIBUTE(Attr)->ValueLength:
		ULONG(PNONRESIDENT_ATTRIBUTE(Attr)->AllocatedSize);
}

ULONG RunLength(PUCHAR Run)
{
	return (*Run & 0xf) + ((*Run >> 4) & 0xf) + 1;
}

ULONGLONG RunCount(PUCHAR Run)
{
	// Get the end index.
	UCHAR Idx = *Run & 0xF;
	ULONGLONG Cnt = 0;

	for (; Idx > 0; Idx--)
		Cnt = (Cnt << 8) + Run[Idx];
	return Cnt;
}

LONGLONG RunLCN(PUCHAR Run)
{
	UCHAR VCNNumEndIdx = *Run & 0xf;
	UCHAR LCNIdxValNum = (*Run >> 4) & 0xf;

	LONGLONG LCN = LCNIdxValNum == 0 ? 0 : CHAR(Run[VCNNumEndIdx +
LCNIdxValNum]);

	for (LONG Idx = VCNNumEndIdx + LCNIdxValNum - 1; Idx > VCNNumEndIdx; Idx--
)
		LCN = (LCN << 8) + Run[Idx];
	return LCN;
}

BOOL FindRun(PNONRESIDENT_ATTRIBUTE Attr, ULONGLONG VCN, PULONGLONG LCN,
PULONGLONG Cnt)
{
	INT Idx;

	if (VCN < Attr->LowVcn || VCN > Attr->HighVcn)
		return FALSE;

	*LCN = 0;
	ULONGLONG Base = Attr->LowVcn;

	PUCHAR Run = PUCHAR(FIXOFFSET(Attr, Attr->RunArrayOffset));

	for (Idx = 0; *Run != 0; Run += RunLength(Run), ++Idx)
	{
		*LCN += RunLCN(Run);
		*Cnt   = RunCount(Run);
```

```
            if (Base <= VCN && VCN < Base + *Cnt)
            {
                  *LCN = RunLCN(Run) == 0 ? 0 : *LCN + VCN - Base;
                  *Cnt    -= ULONG(VCN - Base);
                  return TRUE;
            }
            else
            {
                  Base += *Cnt;
            }
      }
      *Cnt = 0;
      return FALSE;
}

void ZeroExternalAttribute(PNONRESIDENT_ATTRIBUTE Attr, ULONGLONG VCN, ULONG
Cnt)
{
      ULONGLONG LCN, RunClstrCnt;
      ULONG RdCnt, Left;
      ULONG ZOBytes = 0;

      for (Left = Cnt; Left > 0; Left -= RdCnt)
      {
            FindRun(Attr, VCN, &LCN, &RunClstrCnt);
            RdCnt = ULONG(min(RunClstrCnt, Left));

            if(RdCnt != 0)
            {
                  ZOBytes += RdCnt * BytesPerCluster;
                  ZeroLCN(LCN, RdCnt);
                  VCN   += RdCnt;
            }
            else
                  break;
      }
      wprintf(L"Zero data bytes number %d\n", ZOBytes);
}

void ReadExternalAttribute(PNONRESIDENT_ATTRIBUTE Attr, ULONGLONG VCN, ULONG
Cnt, PVOID Buffer)
{
      ULONGLONG LCN, RunClstrCnt;
      ULONG RdCnt, Left;
      PUCHAR DataPtr = PUCHAR(Buffer);

      for (Left = Cnt; Left > 0; Left -= RdCnt)
      {
            FindRun(Attr, VCN, &LCN, &RunClstrCnt);

            RdCnt = ULONG(min(RunClstrCnt, Left));
            ULONG RdBytes = RdCnt * BytesPerCluster;

            if (LCN == 0)
            {
```

```
                  memset(DataPtr, 0, RdBytes);
            }
            else
            {
                  // LCN is physical index of cluster.
                  ReadLCN(LCN, RdCnt, DataPtr);
            }

            // Update virtual cluster index.
            VCN    += RdCnt;
            DataPtr += RdBytes;
      }
}

void ReadAttribute(PATTRIBUTE Attr, PVOID Buffer)
{
      if (Attr->Nonresident == FALSE)
      {
            PRESIDENT_ATTRIBUTE RAttr = PRESIDENT_ATTRIBUTE(Attr);
            memcpy(Buffer, FIXOFFSET(RAttr, RAttr->ValueOffset), RAttr-
>ValueLength);
      }
      else
      {
            PNONRESIDENT_ATTRIBUTE NAttr = PNONRESIDENT_ATTRIBUTE(Attr);
            ReadExternalAttribute(NAttr, 0, ULONG(NAttr->HighVcn) + 1, Buffer);
      }
}

PATTRIBUTE FindAttribute(PFILE_RECORD_HEADER FileRecord, ATTRIBUTE_TYPE Tp,
PWSTR Name)
{
      for (PATTRIBUTE Attr = PATTRIBUTE(FIXOFFSET(FileRecord, FileRecord-
>AttributesOffset));
            Attr->AttributeType != -1;
            Attr = PATTRIBUTE(FIXOFFSET(Attr, Attr->Length)))
      {
            if (Attr->AttributeType == Tp)
            {
                  // This Attribute hasn't name, found return.
                  if (Name == 0 && Attr->NameLength == 0)
                  {
                        return Attr;
                  }

                  if (Name != 0 && wcslen(Name) == Attr->NameLength
                        && _wcsicmp(Name, PWSTR(FIXOFFSET(Attr, Attr-
>NameOffset))) == 0)
                  {
                        return Attr;
                  }
            }
      }
      return NULL;
}
```

```
PATTRIBUTE FindAttributeFileName(PFILE_RECORD_HEADER FileRecord, PWSTR Name)
{
	PATTRIBUTE AttrCp = NULL;
	PFILENAME_ATTRIBUTE FileName;

	for (PATTRIBUTE Attr = PATTRIBUTE(FIXOFFSET(FileRecord, FileRecord-
>AttributesOffset));
		Attr->AttributeType != -1;
		Attr = PATTRIBUTE(FIXOFFSET(Attr, Attr->Length)))
	{
		if (Attr->AttributeType == AttributeFileName)
		{
			// This Attribute has no name, found return.
			if (Name == 0 && Attr->NameLength == 0)
			{
				AttrCp = Attr;
				FileName = PFILENAME_ATTRIBUTE(FIXOFFSET(AttrCp,
					PRESIDENT_ATTRIBUTE(AttrCp)->ValueOffset));

				if(FileName->NameType == 1)
					return AttrCp;
			}

			if (Name != 0 && wcslen(Name) == Attr->NameLength
				&& _wcsicmp(Name, PWSTR(FIXOFFSET(Attr, Attr-
>NameOffset))) == 0)
			{
				AttrCp = Attr;
				FileName = PFILENAME_ATTRIBUTE(FIXOFFSET(AttrCp,
					PRESIDENT_ATTRIBUTE(AttrCp)->ValueOffset));

				if(FileName->NameType == 1)
					return AttrCp;
			}
		}
	}
	return AttrCp;
}

void ReadVCN(PFILE_RECORD_HEADER FileRecord, ATTRIBUTE_TYPE Tp, ULONGLONG VCN,
ULONG Cnt, PVOID Buffer)
{
	PNONRESIDENT_ATTRIBUTE Attr =
PNONRESIDENT_ATTRIBUTE(FindAttribute(FileRecord, Tp, 0));

	if (Attr == 0 || (VCN < Attr->LowVcn || VCN > Attr->HighVcn))
	{
		PATTRIBUTE Attrlist = FindAttribute(FileRecord,
AttributeAttributeList, 0);
		// Will cause a breakpoint exception to occur in the current
process.
		DebugBreak();
	}
	ReadExternalAttribute(Attr, VCN, Cnt, Buffer);
}
```

```
ULONGLONG GetLCN(ULONG Idx)
{
	ULONGLONG VCN = ULONGLONG(Idx) * BytesPerFileRecord / BytesPerCluster;
	PNONRESIDENT_ATTRIBUTE Attr = PNONRESIDENT_ATTRIBUTE(FindAttribute(MFT,
AttributeData, 0));
	ULONGLONG LCN, RunClstrCnt;

	if(FindRun(Attr, VCN, &LCN, &RunClstrCnt) == FALSE)
		return 0;
	return LCN;
}

void ReadFileRecord(ULONG Idx, PFILE_RECORD_HEADER FileRecord)
{
	ULONG ClstrNum = BootBlk.ClustersPerFileRecord;

	if (ClstrNum > 0x80)
		ClstrNum = 1;
	PUCHAR BufPtr = new UCHAR[BytesPerCluster * ClstrNum];

	ULONGLONG VCN = ULONGLONG(Idx) * BytesPerFileRecord / BytesPerCluster;

    ReadVCN(MFT, AttributeData, VCN, ClstrNum, BufPtr);

	LONG FRPerCluster = (BytesPerCluster / BytesPerFileRecord) - 1;
	ULONG FRIdx = FRPerCluster > 0 ? (Idx & FRPerCluster) : 0;

	memcpy(FileRecord, BufPtr + FRIdx * BytesPerFileRecord,
BytesPerFileRecord);
	delete [] BufPtr;
}

void ListDeleted()
{
	ULONG Idx;
	PATTRIBUTE Attr = (PATTRIBUTE)FindAttribute(MFT, AttributeBitmap, 0);
	PUCHAR Bitmap   = new UCHAR[AttributeLengthAllocated(Attr)];

	ReadAttribute(Attr, Bitmap);
	ULONG Num = AttributeLength(FindAttribute(MFT, AttributeData, 0)) /
BytesPerFileRecord;

	fwprintf(pFLog, L"\nMFT Data number %u\n\n", Num);

	PFILE_RECORD_HEADER FileRecord = PFILE_RECORD_HEADER(new
UCHAR[BytesPerFileRecord]);

	for (Idx = 0; Idx < Num; Idx++)
	{
		if (BitSet(Bitmap, Idx))
			continue;

		ReadFileRecord(Idx, FileRecord);
		FixupUpdateSequenceArray(FileRecord);

		if (FileRecord->Ntfs.Type == 'ELIF' && (FileRecord->Flags & 1) == 0)
```

```
            {
                  Attr = (PATTRIBUTE)FindAttributeFileName(FileRecord, 0);
                  if (Attr == 0)
                        continue;

                  PFILENAME_ATTRIBUTE Name = PFILENAME_ATTRIBUTE(FIXOFFSET(Attr,
                        PRESIDENT_ATTRIBUTE(Attr)->ValueOffset));
                  fwprintf(pFLog, L"%10u, %3d, %ws\n", Idx, INT(Name-
>NameLength), Name->Name);
                  // To see the index, file name length and the file name
displayed on the standard output,
                  // uncomment the following line
                  // wprintf(L"%10u %u, %ws\n", Idx, INT(Name->NameLength),
Name->Name);
            }
      }
      delete [] Bitmap;
      delete [] (UCHAR*)FileRecord;
}

void LoadMFT()
{
      BytesPerCluster = BootBlk.SectorsPerCluster * BootBlk.BytesPerSector;
      BytesPerFileRecord = BootBlk.ClustersPerFileRecord < 0x80
            ? BootBlk.ClustersPerFileRecord * BytesPerCluster : (1 << (0x100 -
BootBlk.ClustersPerFileRecord));

      wprintf(L"Cluster Per File Record: %u\n", BootBlk.ClustersPerFileRecord);
      wprintf(L"Bytes Per File Record: %u\n", BytesPerFileRecord);

      MFT = PFILE_RECORD_HEADER(new UCHAR[BytesPerFileRecord]);
      ReadSector(BootBlk.MftStartLcn *
BootBlk.SectorsPerCluster,BytesPerFileRecord / BootBlk.BytesPerSector, MFT);
      FixupUpdateSequenceArray(MFT);
}

VOID UnloadMFT()
{
      // Clean up
      wprintf(L"Unloading MFT...\n");
      delete [] (UCHAR*)MFT;
}

void RemoveFile(ULONG Idx)
{
      PFILE_RECORD_HEADER FileRecord = PFILE_RECORD_HEADER(new
UCHAR[BytesPerFileRecord]);
      UCHAR *ClustersBuf;
      ULONG BufferSize, DataSize, AllocatedSize, Position, LCN, ClstrNum;

      ReadFileRecord(Idx, FileRecord);
      FixupUpdateSequenceArray(FileRecord);

      if (FileRecord->Ntfs.Type != 'ELIF')
      {
            wprintf(L"RemoveFile() - FileRecord->Ntfs.Type != \'ELIF\'...\n");
```

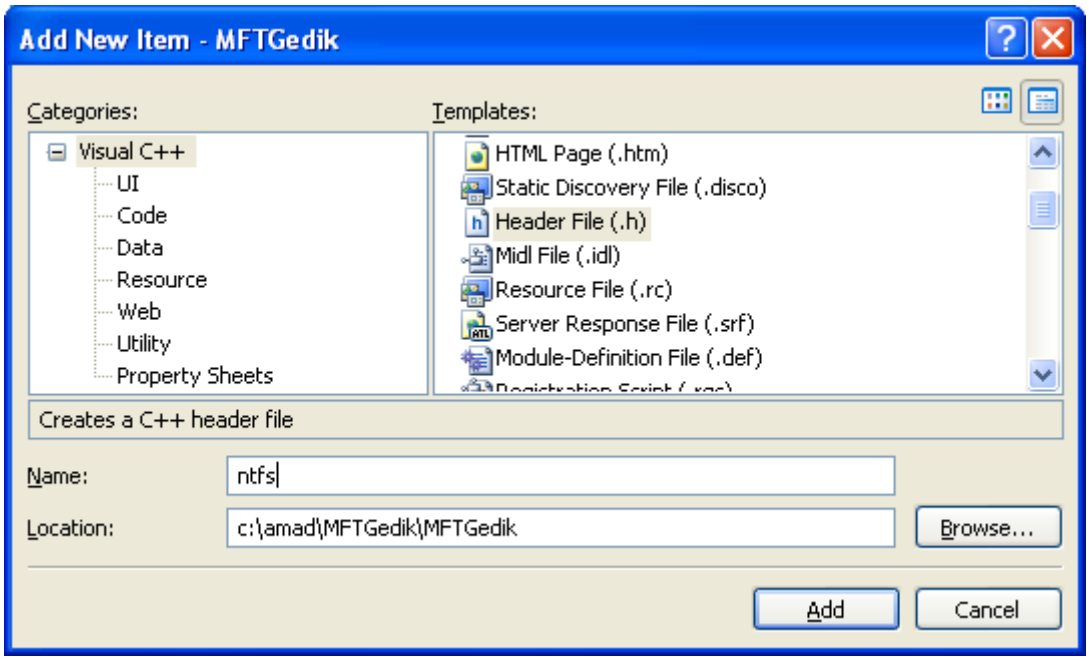
```
            delete [] (UCHAR*)FileRecord;
            return;
      }

      PATTRIBUTE Attr = FindAttribute(FileRecord, AttributeData, 0);

      if (Attr == 0)
      {
            wprintf(L"RemoveFile() - Attr == 0...\n");
            delete [] (UCHAR*)FileRecord;
            return;
      }

      DataSize   = AttributeLength(Attr);
      AllocatedSize = AttributeLengthAllocated(Attr);
      BufferSize   = AllocatedSize > DataSize ? AllocatedSize : DataSize;

      PUCHAR Buffer   = new UCHAR[BufferSize + 1];

      ClstrNum   = BootBlk.ClustersPerFileRecord;

      if (ClstrNum > 0x80)
            ClstrNum = 1;

      ClustersBuf     = new UCHAR[BytesPerCluster * ClstrNum];

      if (Attr->Nonresident == FALSE)
      {
            wprintf(L"This file data is resident!\n");
            PRESIDENT_ATTRIBUTE RAttr = PRESIDENT_ATTRIBUTE(Attr);
            LONG FRPerCluster = (BytesPerCluster / BytesPerFileRecord) - 1;
            ULONG FRIdx = FRPerCluster > 0 ? (Idx & FRPerCluster) : 0;

            Position = (ULONG)(FIXOFFSET(RAttr, RAttr->ValueOffset)) -
(ULONG)FileRecord;
            Position = FRIdx * BytesPerFileRecord + Position;

            if(LCN = (ULONG)GetLCN(Idx), LCN)
            {
                  // Read file record
                  ReadLCN(LCN, ClstrNum, ClustersBuf);

      FixupUpdateSequenceArray((FILE_RECORD_HEADER*)&ClustersBuf[FRIdx *
BytesPerFileRecord]);
                  ZeroSequenceArray((FILE_RECORD_HEADER*)&ClustersBuf[FRIdx *
BytesPerFileRecord]);
                  memset(&ClustersBuf[Position], '0', DataSize);
                  WriteLCN(LCN, ClstrNum, ClustersBuf);
            }
      }
      else
      {
            wprintf(L"This file data is nonresident!\n");
            PNONRESIDENT_ATTRIBUTE NAttr = PNONRESIDENT_ATTRIBUTE(Attr);
            ZeroExternalAttribute(NAttr, 0, ULONG(NAttr->HighVcn) + 1);
      }
```

```
	wprintf(L" Removed file should be succeeded!\n");
	wprintf(L" The file\'s content should be empty!!!\n");
}

void RecoverFile(ULONG Idx, LPCWSTR NewFileName)
{
	PFILE_RECORD_HEADER FileRecord = PFILE_RECORD_HEADER(new
UCHAR[BytesPerFileRecord]);
	ULONG Written, BufferSize, DataSize, AllocatedSize;
	ReadFileRecord(Idx, FileRecord);
	FixupUpdateSequenceArray(FileRecord);

	if (FileRecord->Ntfs.Type != 'ELIF')
	{
		delete [] (UCHAR*)FileRecord;
		wprintf(L"Failed. FileRecord->Ntfs.Type != 'ELIF'\n");
		return;
	}

	PATTRIBUTE Attr = FindAttribute(FileRecord, AttributeData, 0);

	if (Attr == 0)
	{
		delete [] (UCHAR*)FileRecord;
		wprintf(L"Failed, Attr == 0!\n");
		return;
	}

	DataSize   = AttributeLength(Attr);
	AllocatedSize = AttributeLengthAllocated(Attr);
	BufferSize   = AllocatedSize > DataSize ? AllocatedSize : DataSize;
	// Align
	BufferSize    = BufferSize / BootBlk.BytesPerSector *
BootBlk.BytesPerSector + BootBlk.BytesPerSector;

	PUCHAR Buffer = new UCHAR[BufferSize];

	wprintf(L"RecoverFile() - Reading the deleted file data...\n");
	ReadAttribute(Attr, Buffer);

	HANDLE hFile = CreateFile(NewFileName, GENERIC_WRITE, 0, 0, CREATE_ALWAYS,
0, 0);


	if(hFile == INVALID_HANDLE_VALUE)
	{
		wprintf(L"RecoverFile() - CreateFile() failed, error %u\n",
GetLastError());
		exit(1);
	}

	if(WriteFile(hFile, Buffer, DataSize, &Written, 0) == 0)
	{
		wprintf(L"RecoverFile() - WriteFile() failed, error %u\n",
GetLastError());
		exit(1);
```

```
	}

	// Another check
	// Written = the number of byte to be written
	if(Written != DataSize)
		wprintf(L"Writing the file data failed!, error %u\n",
GetLastError());

	CloseHandle(hFile);
	delete [] Buffer;
	delete [] (UCHAR*)FileRecord;
}

int wmain(int argc, WCHAR *argv[])
{
	// Default partition
	WCHAR Drive[] = L"\\\\?\\C:";
	ULONG Read, Idx;
	LPCWSTR OriFileName = L"";
	errno_t ernoFLog1;
	WCHAR *cOption = L"A";

	// Give some info to clueless user
	wprintf(L"Usages:\n");
	wprintf(L"(Default primary partition is C:\\\n");
	wprintf(L"1. %s - Attempting to list the deleted files...\n", argv[0]);
	wprintf(L"	(The deleted files stored in C:\\DeletedFile.txt)\n");
	wprintf(L"2. Finding the deleted file\n");
	wprintf(L"	%s <file_index> <original_file_name>\n", argv[0]);
	wprintf(L"	(The index and file name can be found in
C:\\DeletedFile.txt)\n");
	wprintf(L"	e.g. %s  123546 tergedik.txt\n", argv[0]);
	wprintf(L"3. Removing the deleted file\n");
	wprintf(L"	%s <file_index_to_be_removed>\n", argv[0]);
	wprintf(L"	e.g. %s 123546\n", argv[0]);
	wprintf(L"===Press any key to continue!===\n");

	// Let them read for a while
	getwchar();

	ernoFLog1 = _wfopen_s(&pFLog, L"C:\\DeletedFile.txt", L"w");

	if(ernoFLog1 != 0)
	{
		wprintf(L"_wfopen_s() failed, error %u\n", _get_errno(&ernoFLog1));
		exit(1);
	}

	hVolume = CreateFile(
		Drive,
		GENERIC_READ | GENERIC_WRITE , FILE_SHARE_READ | FILE_SHARE_WRITE,
		0,
		OPEN_EXISTING,
		0,
		0);
```

```
	if( hVolume == INVALID_HANDLE_VALUE)
	{
		wprintf(L"CreateFile() failed, error %u\n", GetLastError());
		exit(1);
	}

	if(ReadFile(hVolume, &BootBlk, sizeof(BOOT_BLOCK), &Read, 0) == 0)
	{
		wprintf(L"ReadFile() failed, error %u\n", GetLastError());
		exit(1);
	}

	wprintf(L"Read volume should succeeded...\n");

	LoadMFT();

	wprintf(L"Load MFT succeed:\n");
	wprintf(L" Bytes Per Sector:Sectors Per Cluster:Clusters Per FileRecord--
>");
	wprintf(L" %u:%u:%u.\n", BootBlk.BytesPerSector,
BootBlk.SectorsPerCluster, BootBlk.ClustersPerFileRecord);

	wprintf(L"Attempt to list the deleted files.........\n");
	wprintf(L"   Find them in C:\\DeletedFile.txt lol!\n");
	ListDeleted();

	// Attempt to find the deleted file from MFT
	if(argc == 3)
	{
		// Use the index and the original file name for recovery.
		// The recovered file stored under the project's debug folder.
		// The index and original file name can be found in
C:\DeletedFile.txt
		Idx = _wtoi(argv[1]);
		OriFileName = argv[2];
		wprintf(L"Recovered file should be in the projects\'s Debug
folder!\n");
		RecoverFile(Idx, OriFileName);
	}

	// Attempt to delete the 'file' from MFT
	if(argc == 2)
	{
		// Use the index to remove the file
		Idx = _wtoi(argv[1]);
		RemoveFile(Idx);
	}

	// Free up all the resources.
	// Look likes some of the malloc() are not freed lol! Who cares?
	// You can do it...
	UnloadMFT();
	CloseHandle(hVolume);
	fclose(pFLog);
}
```

Next, add a ntfs.h header file to the project.



Then, add the source code.

```
// ntfs.h
// Just a portion of the NTFS types
// A more complete can be found in reactos.org
// source code repsitory or other Linux/Unix source code
// repo or at http://www.ntfs-3g.org/
typedef struct {
    ULONG Type;
    USHORT UsaOffset;
    USHORT UsaCount;
    USN Usn;
} NTFS_RECORD_HEADER, *PNTFS_RECORD_HEADER;

typedef struct {
    NTFS_RECORD_HEADER Ntfs;
    USHORT SequenceNumber;
    USHORT LinkCount;
    USHORT AttributesOffset;
      // 0x0001 = InUse, 0x0002 = Directory
    USHORT Flags;
    ULONG BytesInUse;
    ULONG BytesAllocated;
    ULONGLONG BaseFileRecord;
    USHORT NextAttributeNumber;
} FILE_RECORD_HEADER, *PFILE_RECORD_HEADER;

typedef enum {
    AttributeStandardInformation = 0x10,
    AttributeAttributeList = 0x20,
    AttributeFileName = 0x30,
```

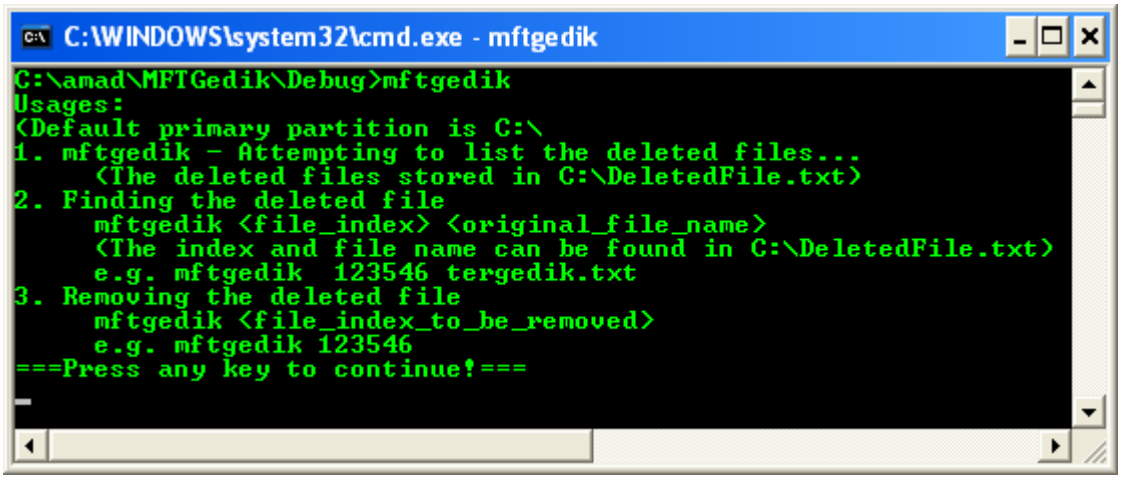
```
    AttributeObjectId = 0x40,
    AttributeSecurityDescriptor = 0x50,
    AttributeVolumeName = 0x60,
    AttributeVolumeInformation = 0x70,
    AttributeData = 0x80,
    AttributeIndexRoot = 0x90,
    AttributeIndexAllocation = 0xA0,
    AttributeBitmap = 0xB0,
    AttributeReparsePoint = 0xC0,
    AttributeEAInformation = 0xD0,
    AttributeEA = 0xE0,
    AttributePropertySet = 0xF0,
    AttributeLoggedUtilityStream = 0x100
} ATTRIBUTE_TYPE, *PATTRIBUTE_TYPE;

typedef struct {
    ATTRIBUTE_TYPE AttributeType;
    ULONG Length;
    BOOLEAN Nonresident;
    UCHAR NameLength;
    USHORT NameOffset;
      // 0x0001 = Compressed
    USHORT Flags;
    USHORT AttributeNumber;
} ATTRIBUTE, *PATTRIBUTE;

typedef struct {
    ATTRIBUTE Attribute;
    ULONG ValueLength;
    USHORT ValueOffset;
      // 0x0001 = Indexed
    USHORT Flags;
} RESIDENT_ATTRIBUTE, *PRESIDENT_ATTRIBUTE;

typedef struct {
    ATTRIBUTE Attribute;
    ULONGLONG LowVcn;
    ULONGLONG HighVcn;
    USHORT RunArrayOffset;
    UCHAR CompressionUnit;
    UCHAR AlignmentOrReserved[5];
    ULONGLONG AllocatedSize;
    ULONGLONG DataSize;
    ULONGLONG InitializedSize;
      // Only when compressed
    ULONGLONG CompressedSize;
} NONRESIDENT_ATTRIBUTE, *PNONRESIDENT_ATTRIBUTE;

typedef struct {
    ULONGLONG CreationTime;
    ULONGLONG ChangeTime;
    ULONGLONG LastWriteTime;
    ULONGLONG LastAccessTime;
    ULONG FileAttributes;
    ULONG AlignmentOrReservedOrUnknown[3];
    ULONG QuotaId;                  // NTFS 3.0 only
```

```
    ULONG SecurityId;          // NTFS 3.0 only
    ULONGLONG QuotaCharge;     // NTFS 3.0 only
    USN Usn;                           // NTFS 3.0 only
} STANDARD_INFORMATION, *PSTANDARD_INFORMATION;

typedef struct {
    ATTRIBUTE_TYPE AttributeType;
    USHORT Length;
    UCHAR NameLength;
    UCHAR NameOffset;
    ULONGLONG LowVcn;
    ULONGLONG FileReferenceNumber;
    USHORT AttributeNumber;
    USHORT AlignmentOrReserved[3];
} ATTRIBUTE_LIST, *PATTRIBUTE_LIST;

typedef struct {
    ULONGLONG DirectoryFileReferenceNumber;
    ULONGLONG CreationTime;    // Saved when filename last changed
    ULONGLONG ChangeTime;      // ditto
    ULONGLONG LastWriteTime;   // ditto
    ULONGLONG LastAccessTime;  // ditto
    ULONGLONG AllocatedSize;   // ditto
    ULONGLONG DataSize;        // ditto
    ULONG FileAttributes;      // ditto
    ULONG AlignmentOrReserved;
    UCHAR NameLength;
    UCHAR NameType;            // 0x01 = Long, 0x02 = Short
    WCHAR Name[1];
} FILENAME_ATTRIBUTE, *PFILENAME_ATTRIBUTE;

typedef struct {
    GUID ObjectId;
    union {
        struct {
            GUID BirthVolumeId;
            GUID BirthObjectId;
            GUID DomainId;
        } ;
        UCHAR ExtendedInfo[48];
    };
} OBJECTID_ATTRIBUTE, *POBJECTID_ATTRIBUTE;

typedef struct {
    ULONG Unknown[2];
    UCHAR MajorVersion;
    UCHAR MinorVersion;
    USHORT Flags;
} VOLUME_INFORMATION, *PVOLUME_INFORMATION;

typedef struct {
    ULONG EntriesOffset;
    ULONG IndexBlockLength;
    ULONG AllocatedSize;
    ULONG Flags;          // 0x00 = Small directory, 0x01 = Large directory
} DIRECTORY_INDEX, *PDIRECTORY_INDEX;
```

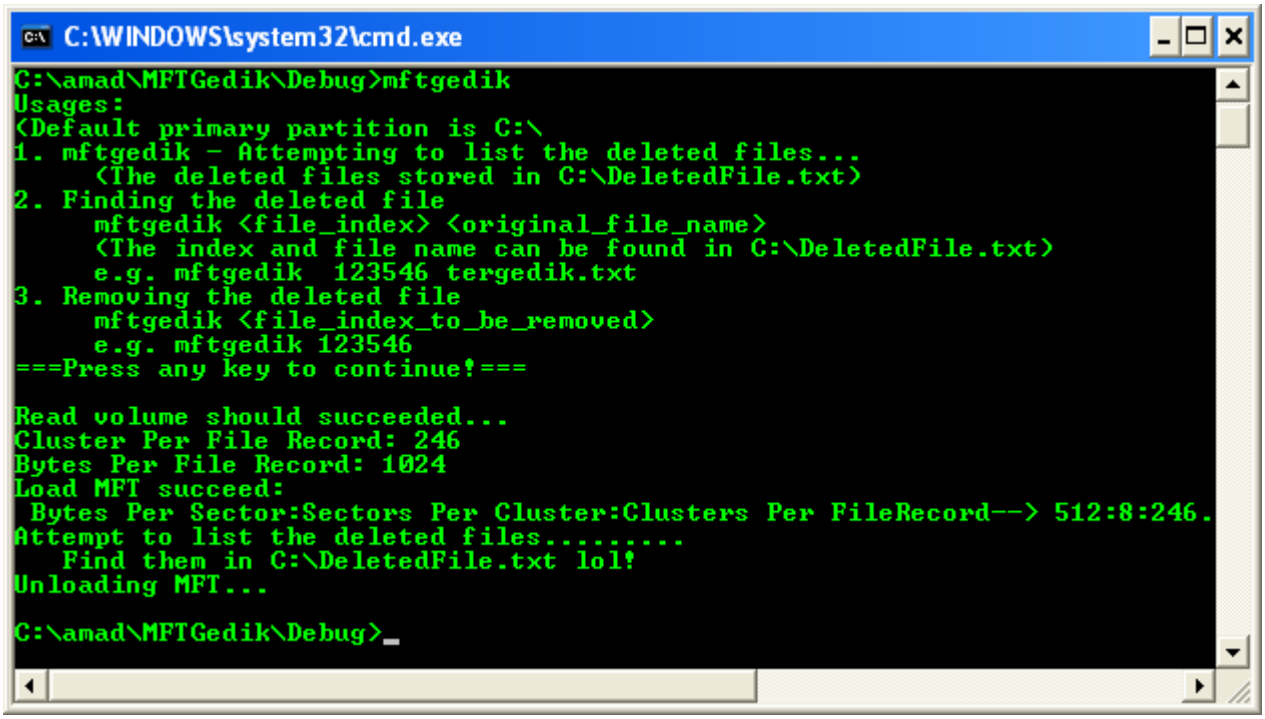
```
typedef struct {
    ULONGLONG FileReferenceNumber;
    USHORT Length;
    USHORT AttributeLength;
    ULONG Flags;            // 0x01 = Has trailing VCN, 0x02 = Last entry
    // FILENAME_ATTRIBUTE Name;
    // ULONGLONG Vcn;       // VCN in IndexAllocation of earlier entries
} DIRECTORY_ENTRY, *PDIRECTORY_ENTRY;

typedef struct {
    ATTRIBUTE_TYPE Type;
    ULONG CollationRule;
    ULONG BytesPerIndexBlock;
    ULONG ClustersPerIndexBlock;
    DIRECTORY_INDEX DirectoryIndex;
} INDEX_ROOT, *PINDEX_ROOT;

typedef struct {
    NTFS_RECORD_HEADER Ntfs;
    ULONGLONG IndexBlockVcn;
    DIRECTORY_INDEX DirectoryIndex;
} INDEX_BLOCK_HEADER, *PINDEX_BLOCK_HEADER;

typedef struct {
    ULONG ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    UCHAR ReparseData[1];
} REPARSE_POINT, *PREPARSE_POINT;

typedef struct {
    ULONG EaLength;
    ULONG EaQueryLength;
} EA_INFORMATION, *PEA_INFORMATION;

typedef struct {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
    // UCHAR EaData[];
} EA_ATTRIBUTE, *PEA_ATTRIBUTE;

typedef struct {
    WCHAR AttributeName[64];
    ULONG AttributeNumber;
    ULONG Unknown[2];
    ULONG Flags;
    ULONGLONG MinimumSize;
    ULONGLONG MaximumSize;
} ATTRIBUTE_DEFINITION, *PATTRIBUTE_DEFINITION;

#pragma pack(push, 1)
```

```
typedef struct {
    UCHAR Jump[3];
    UCHAR Format[8];
    USHORT BytesPerSector;
    UCHAR SectorsPerCluster;
    USHORT BootSectors;
    UCHAR Mbz1;
    USHORT Mbz2;
    USHORT Reserved1;
    UCHAR MediaType;
    USHORT Mbz3;
    USHORT SectorsPerTrack;
    USHORT NumberOfHeads;
    ULONG PartitionOffset;
    ULONG Reserved2[2];
    ULONGLONG TotalSectors;
    ULONGLONG MftStartLcn;
    ULONGLONG Mft2StartLcn;
    ULONG ClustersPerFileRecord;
    ULONG ClustersPerIndexBlock;
    ULONGLONG VolumeSerialNumber;
    UCHAR Code[0x1AE];
    USHORT BootSignature;
} BOOT_BLOCK, *PBOOT_BLOCK;

#pragma pack(pop)
```

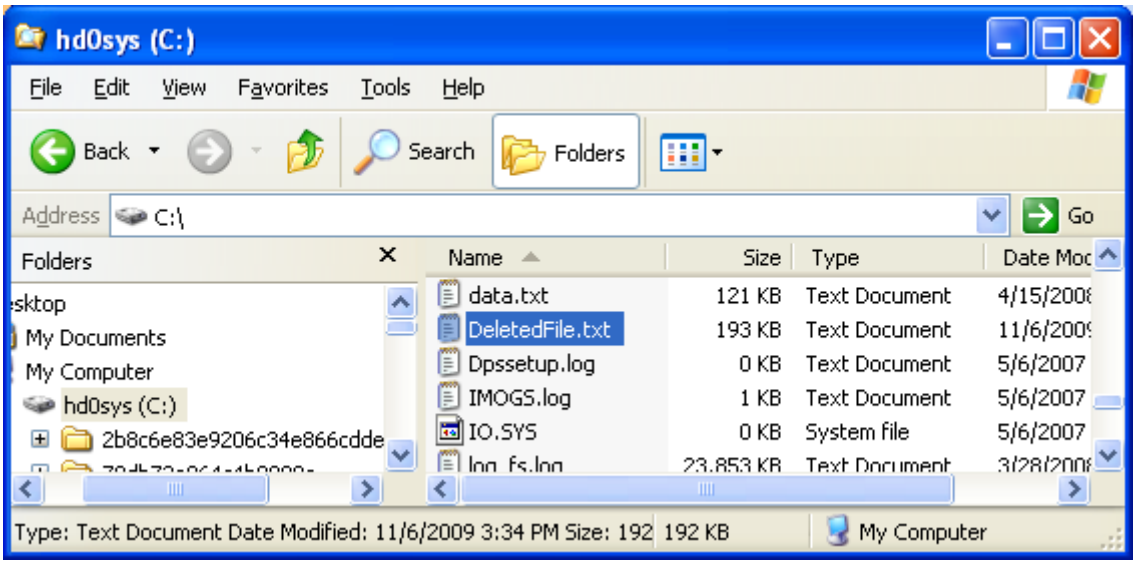
Build and run the project. The following screenshot is an output sample.



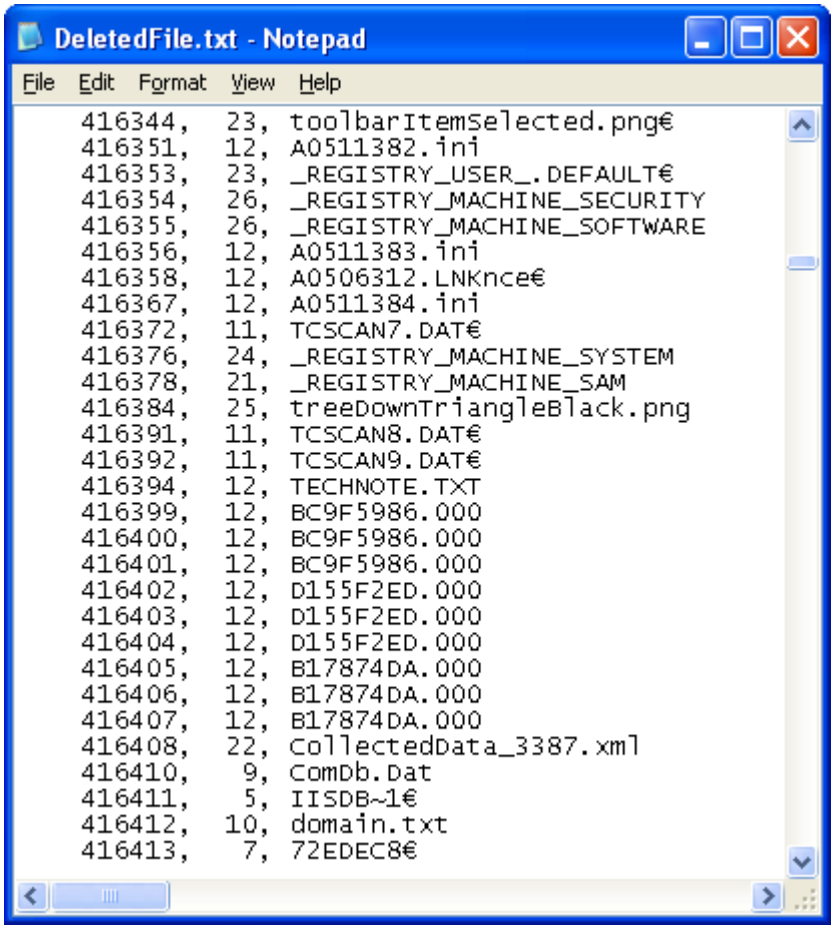
When pressing any key, the deleted files (index, file size and file name) are stored in the DeletedFile.txt



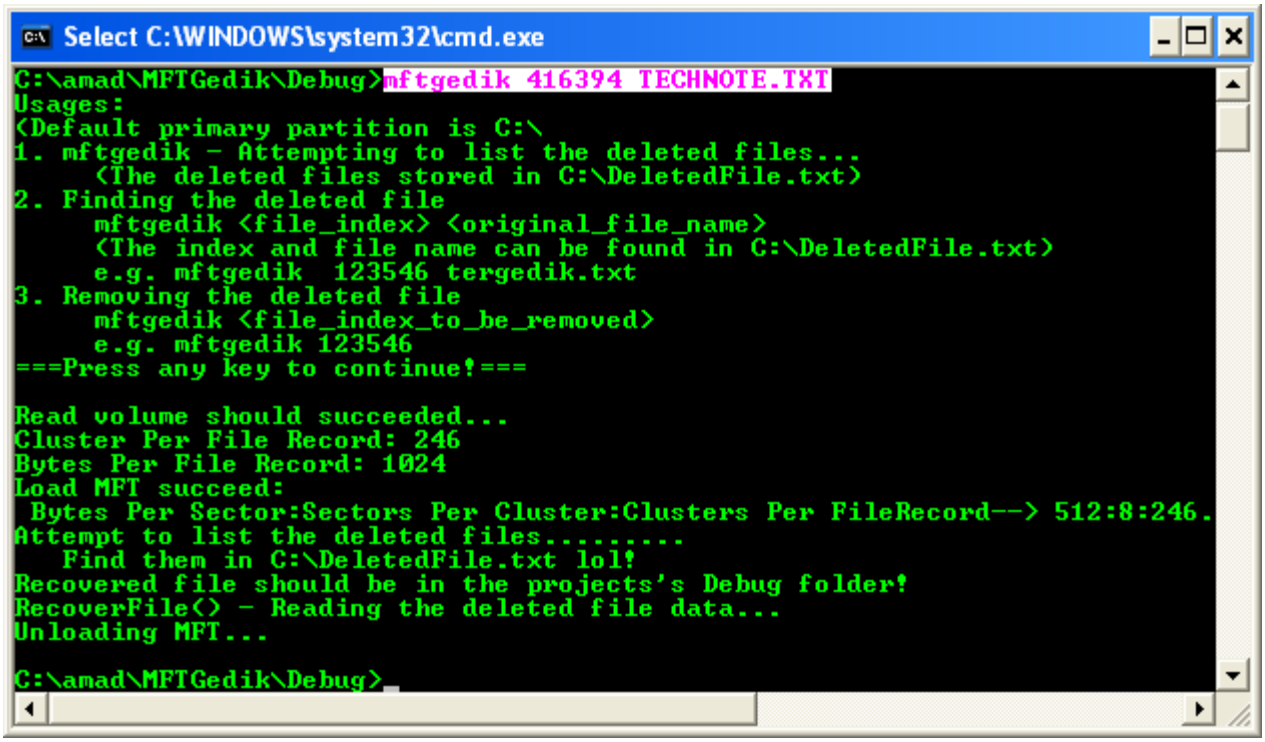
Next, open the DeletedFile.txt.



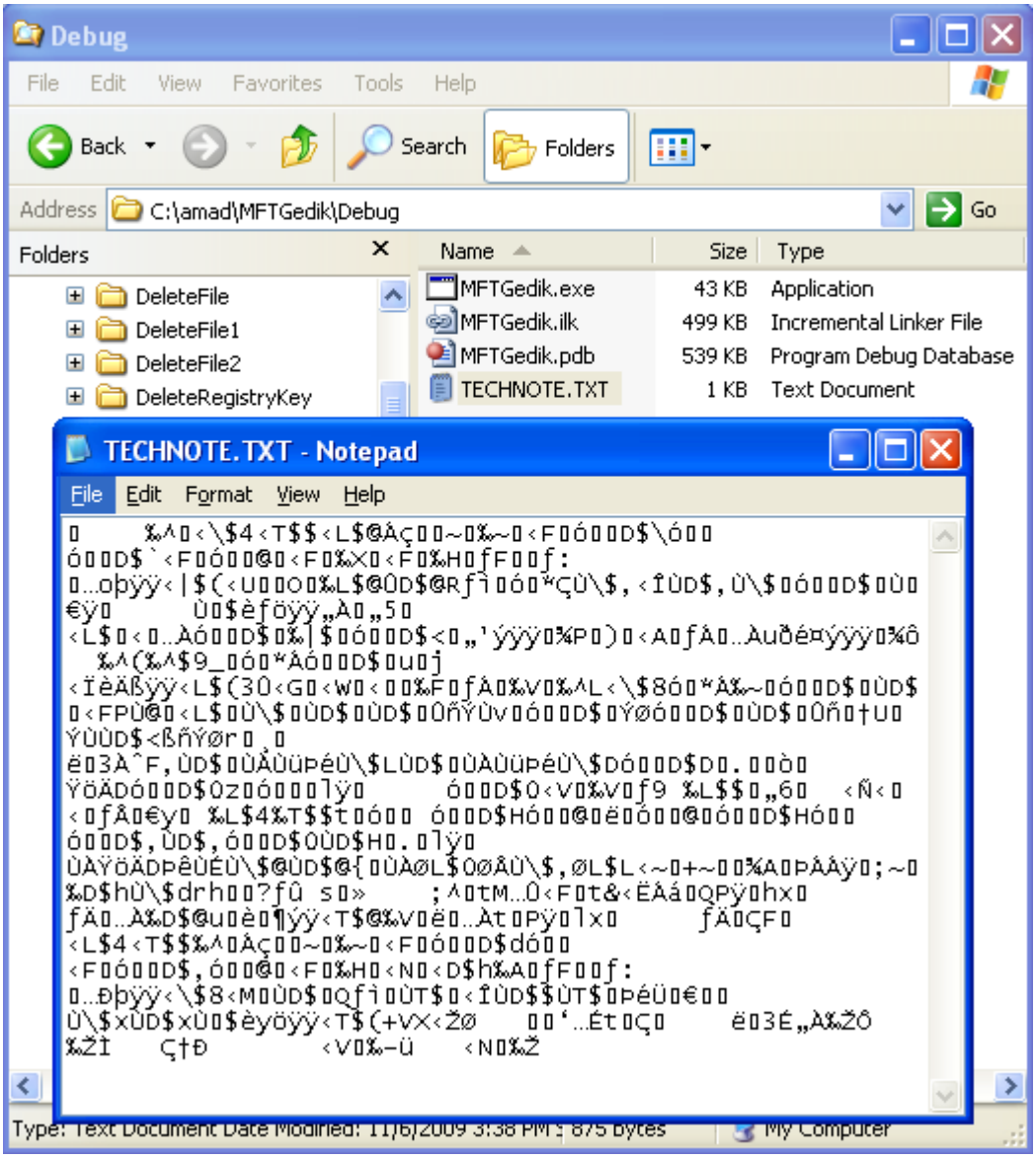
Then let try to recover a file.



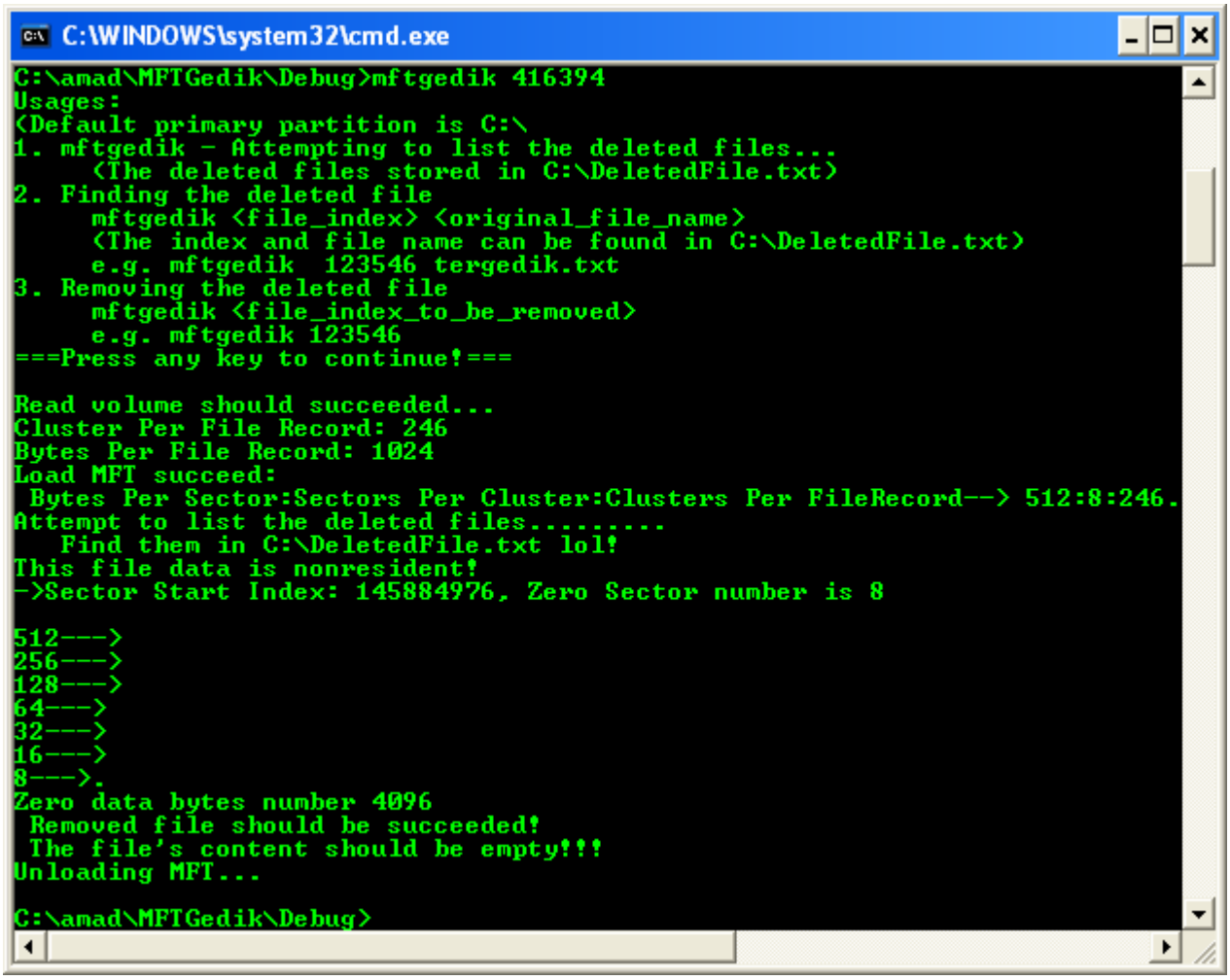
From the DeletedFile.txt, we choose one file. In this case, TECHNOTE.TXT (with index 416394). Then we re-run the program with the index and file name as the arguments.



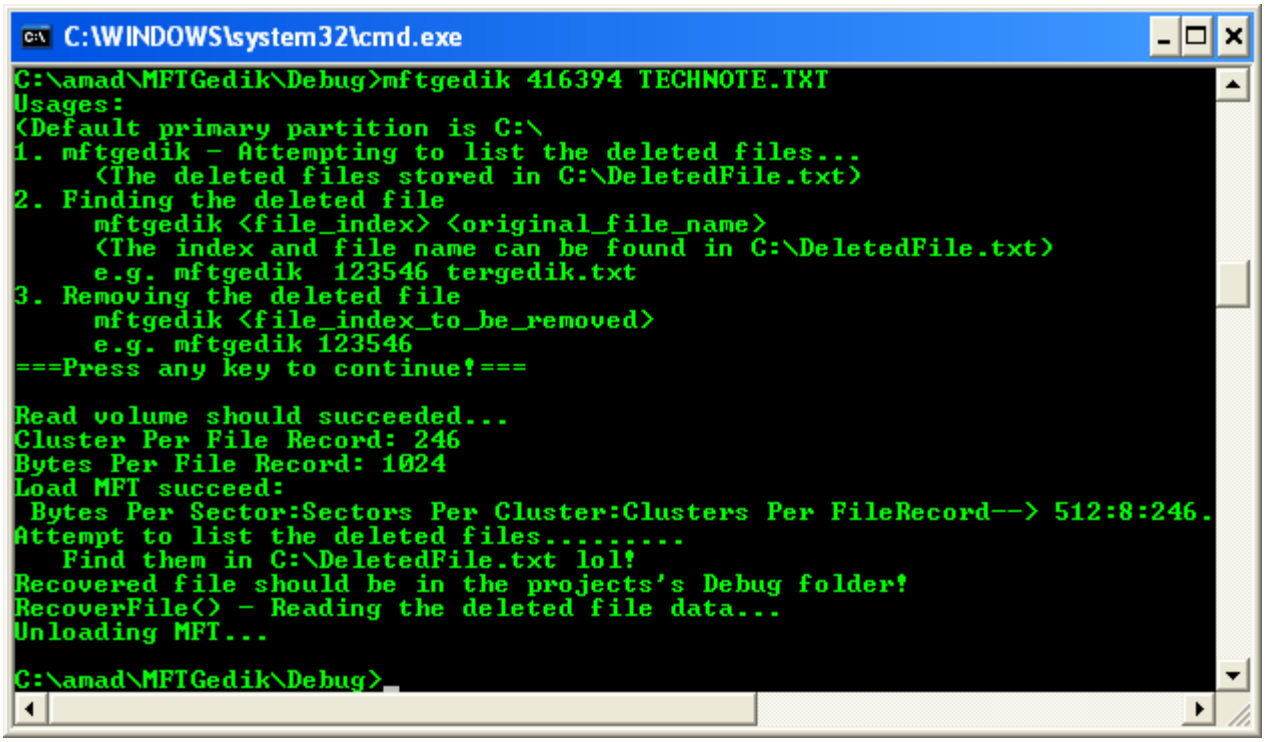
The recovered file should be stored under the project's Debug folder.



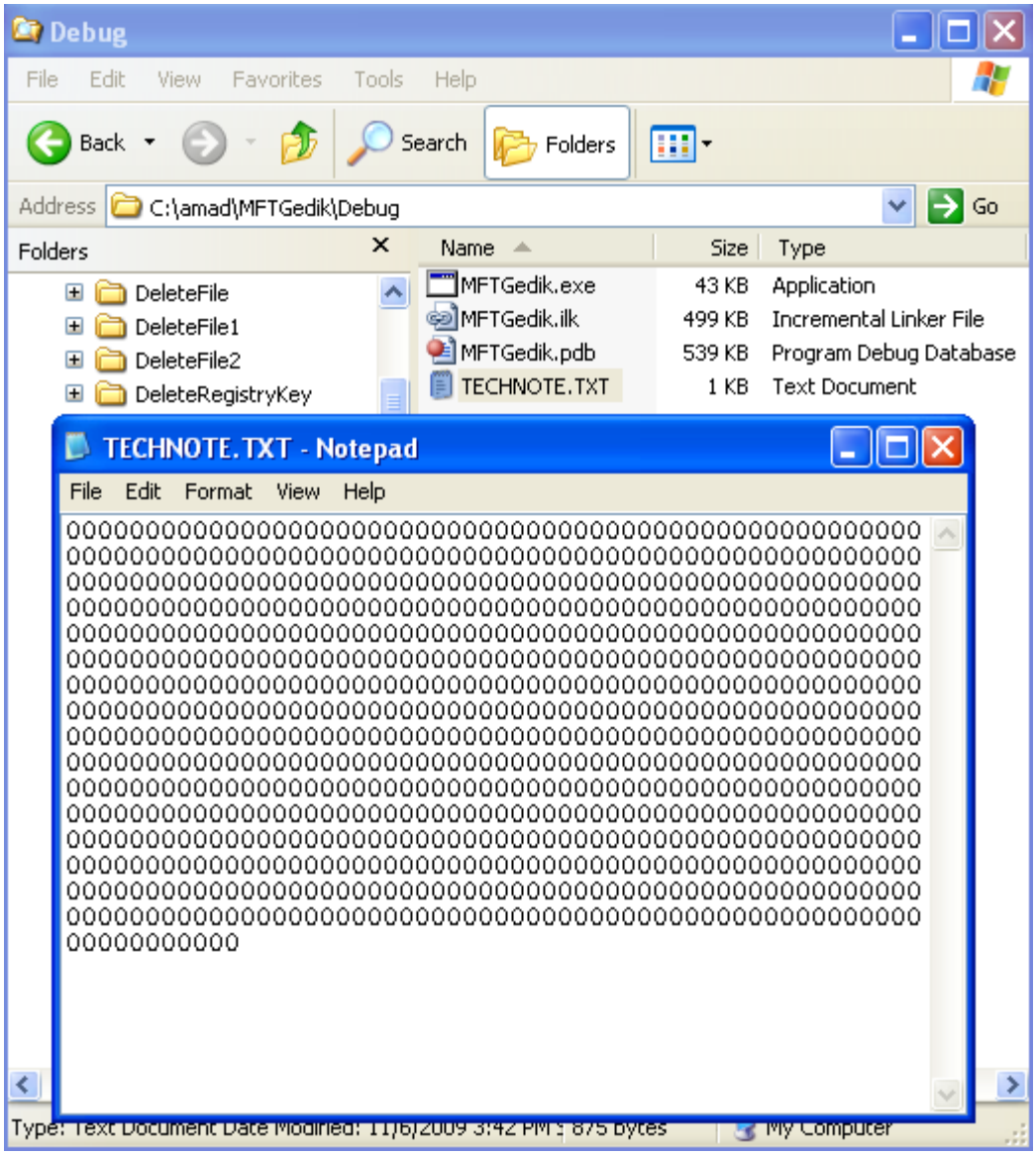
The next task is to delete the file reference in MFT. By using the index we re-run the program with the index as an argument.



Then, let verify the previous task. Re-run the program to recover the same file as done previously.



Re-open the recovered file. As shown in the following Figure, the files content is filled with 0. Just zeroing out the 'content' huh?



More information for Windows MFT can be found in the following links:

1. NTFS On-Disk Structures - Visual Basic NTFS Programmer's Guide by Alex Ionescu (pdf)
2. NTFS Documentation - the Linux NTFS driver by Richard Russon and Yuval Fledel (pdf)
3. ReactOS – The Windows clone project.
4. Windows ® Internals, Fifth Edition
5. NTFS On-Disk Structures – C code and older version compared to no. 1 (pdf).

Windows Master Boot Record (MBR)

There is 'no' information to extract or manipulate the Windows MBR data in MSDN. Many people use Hex editor to view the MBR. Most of the headers dealing with MBR are available in Windows Driver Kit (WDK). However, there are many headers and libraries created by third party and individual for Windows MBR. For Windows 7 and Server 2008 R2,

FSCTL_GET_BOOT_AREA_INFO control code can be used together with BOOT_AREA_INFO structure to retrieve the locations of boot sectors for a volume. Hopefully, the libraries will be expanded for more features in the future. The following list redirects you for more information on Windows MBR.

1. CodeProject: How to develop your own Boot Loader
2. Official WDK and Developer Tools Home
3. Windows Driver Kit (WDK) Documentation Blog
4. TestDisk: A very nice multi OS data recovery – for MBR and MFT.

Volume Management Reference

Volume Management Functions

The following functions are used in volume management.

Function	Description
DefineDosDevice()	Defines, redefines, or deletes MS-DOS device names.
GetDriveType()	Determines whether a disk drive is a removable, fixed, CD-ROM, RAM disk, or network drive.
GetLogicalDrives()	Retrieves a bitmask representing the currently available disk drives.
GetLogicalDriveStrings()	Fills a buffer with strings that specify valid drives in the system.
GetVolumeInformation()	Retrieves information about the file system and volume associated with the specified root directory.
GetVolumeInformationByHandleW()	Retrieves information about the file system and volume associated with the specified file.
QueryDosDevice()	Retrieves information about MS-DOS device names.
SetVolumeLabel()	Sets the label of a file system volume.

The following functions are used with volume mount points (drive letters, volume GUID paths, and mounted folders).

Function	Description
DeleteVolumeMountPoint()	Deletes a drive letter or mounted folder.
FindFirstVolume()	Retrieves the name of a volume on a computer.
FindFirstVolumeMountPoint()	Retrieves the name of a mounted folder on the specified volume.
FindNextVolume()	Continues a volume search started by a call to FindFirstVolume.
FindNextVolumeMountPoint()	Continues a mounted folder search started by a call to FindFirstVolumeMountPoint.
FindVolumeClose()	Closes the specified volume search handle.
FindVolumeMountPointClose()	Closes the specified mounted folder search handle.
GetVolumeNameForVolumeMountPoint()	Retrieves a volume GUID path for the volume that is associated with the specified volume mount point (drive letter, volume GUID path, or mounted folder).
GetVolumePathName()	Retrieves the mounted folder that is associated with the specified volume.
GetVolumePathNamesForVolumeName()	Retrieves a list of drive letters and volume GUID paths for the specified volume.

SetVolumeMountPoint()	Associates a volume with a drive letter or a directory on another volume.

Volume Management Control Codes

The following control codes are used in volume management.

Control code	Operation
FSCTL_DISMOUNT_VOLUME	Dismounts a volume.
FSCTL_EXTEND_VOLUME	Increases the size of a mounted volume.
FSCTL_GET_BOOT_AREA_INFO	Retrieves the locations of boot sectors for a volume.
FSCTL_GET_NTFS_VOLUME_DATA	Retrieves information about the specified NTFS file system volume.
FSCTL_IS_VOLUME_MOUNTED	Determines whether the specified volume is mounted, or if the specified file or directory is on a mounted volume.
FSCTL_LOCK_VOLUME	Locks a volume.
FSCTL_QUERY_FILE_SYSTEM_RECOGNITION	Queries for file system recognition information on a volume.
FSCTL_READ_FROM_PLEX	Reads from the specified plex.
FSCTL_SHRINK_VOLUME	Signals that the volume is to be prepared to perform the shrink operation, the shrink operation is to be committed, or the shrink operation is to be terminated.
FSCTL_UNLOCK_VOLUME	Unlocks a volume.
IOCTL_VOLUME_GET_GPT_ATTRIBUTES	Retrieves attributes associated with a storage volume.
IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS	Retrieves the physical location of the specified volume on one or more disks.
IOCTL_VOLUME_IS_CLUSTERED	Determines whether a volume is clustered.
IOCTL_VOLUME_OFFLINE	Takes a volume offline.
IOCTL_VOLUME_ONLINE	Brings a volume online.

The following control codes are used with change journals.

Value	Meaning
FSCTL_CREATE_USN_JOURNAL	Creates a change journal stream on a target volume or modifies an existing change journal stream.
FSCTL_DELETE_USN_JOURNAL	Deletes a change journal on a volume or awaits notification of deletion of a change journal.
FSCTL_ENUM_USN_DATA	Creates an enumeration that lists the change journal entries between two specified boundaries.
FSCTL_MARK_HANDLE	Marks a specified file or directory and its change journal record with information about changes to that file or directory.
FSCTL_QUERY_USN_JOURNAL	Queries for information on the current change journal, its records, and its capacity.
FSCTL_READ_FILE_USN_DATA	Retrieves the change-journal information for the specified file or directory.
FSCTL_READ_USN_JOURNAL	Returns to the calling process the set of change journal records between two specified USN values.
FSCTL_WRITE_USN_CLOSE_RECORD	Generates a record in the change journal stream for the input file. This record will have the USN_REASON_CLOSE flag.

The following are defragmentation control codes.

Value	Meaning
FSCTL_GET_RETRIEVAL_POINTER_BASE	Returns the sector offset to the first logical cluster number of the file system relative to the start of the volume.
FSCTL_GET_RETRIEVAL_POINTERS	Gets information about the cluster use of a file.
FSCTL_GET_VOLUME_BITMAP	Gets a bitmap of cluster allocation.
FSCTL_MOVE_FILE	Moves all or part of a file from one set of clusters to another within a volume.

Volume Management Structures

The following structures are used in volume management.

Structure	Meaning
BOOT_AREA_INFO	Contains the output for the FSCTL_GET_BOOT_AREA_INFO control code.
CREATE_USN_JOURNAL_DATA	Contains information that describes a change journal.
DELETE_USN_JOURNAL_DATA	Contains information on the deletion of an NTFS file system change journal using the FSCTL_DELETE_USN_JOURNAL control code.
FILE_SYSTEM_RECOGNITION_INFORMATION	Contains file system recognition information retrieved by the FSCTL_QUERY_FILE_SYSTEM_RECOGNITION control code.
FILE_SYSTEM_RECOGNITION_STRUCTURE	Contains the on-disk file system recognition information stored in the volume's boot sector (logical disk sector zero). This is an internally-defined data structure not available in a public header and is provided here for file system developers who want to take advantage of file system recognition.
MARK_HANDLE_INFO	Contains information that is used to mark a specified file or directory, and its change journal record with data about changes. It is used by the FSCTL_MARK_HANDLE control code.
MOVE_FILE_DATA	Contains input data for the FSCTL_MOVE_FILE control code.
MFT_ENUM_DATA	Contains information defining the boundaries for and starting place of an enumeration of change journal records. It is used by the FSCTL_ENUM_USN_DATA control code.
NTFS_VOLUME_DATA_BUFFER	Represents volume data. This structure is passed to the FSCTL_GET_NTFS_VOLUME_DATA control code.
PLEX_READ_DATA_REQUEST	Indicates the range of the read operation to perform and the plex from which to read.
READ_USN_JOURNAL_DATA	Contains information defining a set of change journal records to return to the calling process. It is used by the

	FSCTL_QUERY_USN_JOURNAL and FSCTL_READ_USN_JOURNAL control codes.
RETRIEVAL_POINTER_BASE	Contains the output for the FSCTL_GET_RETRIEVAL_POINTER_BASE control code.
RETRIEVAL_POINTERS_BUFFER	Contains the output for the FSCTL_GET_RETRIEVAL_POINTERS control code.
SHRINK_VOLUME_INFORMATION	Specifies the volume shrink operation to perform.
STARTING_LCN_INPUT_BUFFER	Contains the starting LCN to the FSCTL_GET_VOLUME_BITMAP control code.
STARTING_VCN_INPUT_BUFFER	Contains the starting VCN to the FSCTL_GET_RETRIEVAL_POINTERS control code.
USN_JOURNAL_DATA	Represents a change journal, its records, and its capacity. This structure is the output buffer for the FSCTL_QUERY_USN_JOURNAL control code.
USN_RECORD	Contains the information for a change journal version 2.0 record. Applications should not attempt to work with change journal versions earlier than 2.0.
VOLUME_BITMAP_BUFFER	Represents the occupied and available clusters on a disk. This structure is the output buffer for the FSCTL_GET_VOLUME_BITMAP control code.
VOLUME_DISK_EXTENTS	Represents a physical location on a disk. It is the output buffer for the IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS control code.
VOLUME_GET_GPT_ATTRIBUTES_INFORMATION	Contains volume attributes retrieved with the IOCTL_VOLUME_GET_GPT_ATTRIBUTES control code.

More related references:

1. Basic Disks and Volumes Technical Reference
2. Dynamic Disks and Volumes Technical Reference
3. NTFS Technical Reference