# Windows Character Mode Application

What do we have in this session?

## Introduction

Windows-based applications consist of executable files and DLLs. Most applications interact with the user through a **graphical user interface (GUI)** or a **character-mode interface**. A running application is known as a process. Each process owns system resources. The threads of a process execute its code.
The following topics describe the creation and usage of DLLs, processes, and threads. However, for this session we will only discuss Character-Mode Applications.

| Overview | Description |
|---|---|
| Character-Mode Applications | Character-mode applications do not provide their own graphical user interface. Instead, they interact with consoles. |
| Dynamic-Link Libraries | DLLs are executable modules that contain functions and data. DLLs provide a way to modularize applications so they can be loaded, updated, and reused more easily. |
| Process Status Helper | The process status helper functions make it easier for you to obtain information about processes and device drivers. |
| Processes and | A thread is the basic unit to which the operating system allocates processor |

| | |
|---|---|
| Threads | time. A process is an executing application that consists of one or more threads. |
| Services | A service is an application that conforms to the interface rules of the Service Control Manager. Services can execute even when no user is logged on. |
| Synchronization | Threads can use synchronization functions to coordinate access to a resource. |
| Tool Help Library | The functions provided by the tool help library make it easier for you to obtain information about currently executing applications. |
| Window Stations and Desktops | A desktop is a securable object contained within a window station. A desktop has a logical display surface and contains user interface objects such as windows, menus, and hooks. Each desktop is associated with a thread and can be used to create and manage windows. |

## Character-Mode Applications

In this chapter we will concentrate on the character-mode applications. Consoles manage input and output (I/O) for character-mode applications which are applications that do not provide their own graphical user interface.
The console functions enable different levels of access to a console. The high-level console I/O functions enable an application to read from standard input to retrieve keyboard input stored in a console's input buffer. The functions also enable an application to write to standard output or standard error to display text in the console's screen buffer. The high-level functions also support redirection of standard handles and control of console modes for different I/O functionality. The low-level console I/O functions enable applications to receive detailed input about keyboard and mouse events, as well as events involving user interactions with the console window. The low-level functions also enable greater control of output to the screen.

## Intro to Character-Mode Applications

Consoles provide high-level support for simple character-mode applications that interact with the user by using functions that read from standard input and write to standard output or standard error. Consoles also provide sophisticated low-level support that gives direct access to a console's screen buffer and that enables applications to receive extended input information (such as mouse input).

## Consoles

A console is an interface that provides I/O to character-mode applications. This processor-independent mechanism makes it easy to port existing character-mode applications or to create new character-mode tools and applications.
A console consists of an input buffer and one or more screen buffers. The input buffer contains a queue of input records, each of which contains information about an input event. The input queue always includes key-press and key-release events. It can also include mouse events (pointer movements and button presses and releases) and events during which user actions affect the size of the active screen buffer. A screen buffer is a two-dimensional array of character and color data for output in a console window. Any number of processes can share a console.
We won't dive the details for this topic. For more info please find it in MSDN.

## Using the Console: Program Examples

However, we will try all the given program examples which demonstrate how to use the console functions:

1. Using the high-level input and output functions
2. Reading and writing blocks of characters and attributes
3. Reading input buffer events
4. Clearing the screen
5. Scrolling a screen buffer's window
6. Scrolling a screen buffer's contents
7. Registering a control handler function

## Using the High-Level Input and Output Functions Example

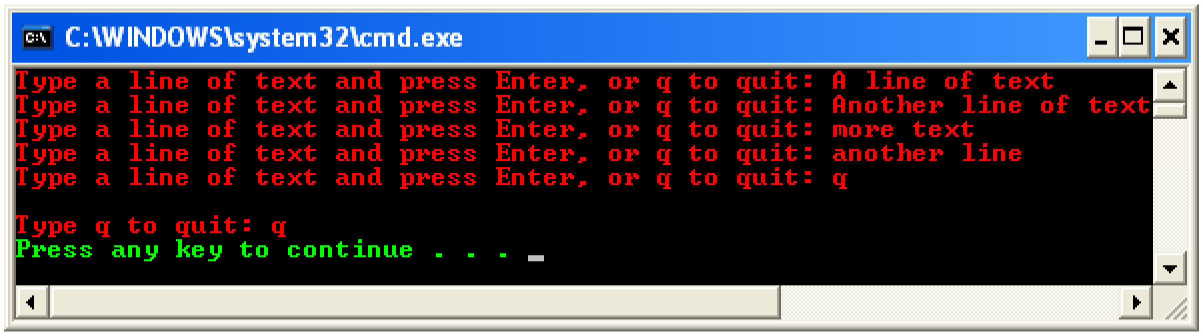
The following example uses the high-level console I/O functions for console I/O. The example assumes that the default I/O modes are in effect initially for the first calls to the ReadFile() and WriteFile() functions. Then the input mode is changed to turn off line input mode and echo input mode for the second calls to ReadFile() and WriteFile(). The SetConsoleTextAttribute() function is used to set the colors in which subsequently written text will be displayed. Before exiting, the program restores the original console input mode and color attributes.
The example's NewLine() function is used when line input mode is disabled. It handles carriage returns by moving the cursor position to the first cell of the next row. If the cursor is already in the last row of the console screen buffer, the contents of the console screen buffer are scrolled up one line.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```
#include <windows.h>

// Prototypes
void NewLine(void);
void ScrollScreenBuffer(HANDLE, INT);
//Global variables
HANDLE hStdout, hStdin;
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;

int wmain(int argc, WCHAR **argv)
{
    LPSTR lpszPrompt1 = "Type a line of text and press Enter, or q to quit:
";
    LPSTR lpszPrompt2 = "Type q to quit: ";
    CHAR chBuffer[256];
    DWORD cRead, cWritten, fdwMode, fdwOldMode;
    WORD wOldColorAttrs;

    // Get handles to STDIN and STDOUT
    hStdin = GetStdHandle(STD_INPUT_HANDLE);
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hStdin == INVALID_HANDLE_VALUE || hStdout == INVALID_HANDLE_VALUE)
    {
        MessageBox(NULL, TEXT("GetStdHandle"), TEXT("Console Error"), MB_OK);
        return 1;
    }

    // Save the current text colors
    if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"), TEXT("Console
Error"), MB_OK);
        return 1;
    }

    wOldColorAttrs = csbiInfo.wAttributes;

    // Set the text attributes to draw red text on black background
    if (!SetConsoleTextAttribute(hStdout, FOREGROUND_RED |
FOREGROUND_INTENSITY))
    {
        MessageBox(NULL, TEXT("SetConsoleTextAttribute"), TEXT("Console
Error"), MB_OK);
        return 1;
    }

    // Write to STDOUT and read from STDIN by using the default
    // modes. Input is echoed automatically, and ReadFile
    // does not return until a carriage return is typed.
    //
    // The default input modes are line, processed, and echo.
    // The default output modes are processed and wrap at EOL
    while (1)
    {
        if (!WriteFile(
            hStdout,               // output handle
            lpszPrompt1,           // prompt string
            lstrlenA(lpszPrompt1), // string length
            &cWritten,             // bytes written
            NULL) )                // not overlapped
        {
            MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),
MB_OK);
            return 1;
        }

        if (! ReadFile(
            hStdin,    // input handle
            chBuffer,  // buffer to read into
            255,       // size of buffer
            &cRead,    // actual bytes read
            NULL) )    // not overlapped
        break;
        if (chBuffer[0] == 'q') break;
    }

    // Turn off the line input and echo input modes
    if (!GetConsoleMode(hStdin, &fdwOldMode))
    {
       MessageBox(NULL, TEXT("GetConsoleMode"), TEXT("Console Error"),
MB_OK);
       return 1;
    }

    fdwMode = fdwOldMode & ~(ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT);
    if (!SetConsoleMode(hStdin, fdwMode))
    {
       MessageBox(NULL, TEXT("SetConsoleMode"), TEXT("Console Error"),
MB_OK);
       return 1;
    }

    // ReadFile returns when any input is available.
    // WriteFile is used to echo input.
    NewLine();

    while (1)
    {
        if (! WriteFile(
            hStdout,               // output handle
            lpszPrompt2,           // prompt string
            lstrlenA(lpszPrompt2), // string length
            &cWritten,             // bytes written
            NULL) )                // not overlapped
        {
            MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),
MB_OK);
            return 1;
        }
```

```
            // If not read from standard input, max 1 byte to read
        if (!ReadFile(hStdin, chBuffer, 1, &cRead, NULL))
            break;
            // If carriage return
        if (chBuffer[0] == '\r')
            NewLine();
            // If not write to standard output
        else if (!WriteFile(hStdout, chBuffer, cRead, &cWritten, NULL))
                  break;
        else
            NewLine();
            // If character 'q'
        if (chBuffer[0] == 'q')
                  break;
    }

    // Restore the original console mode.
    SetConsoleMode(hStdin, fdwOldMode);

    // Restore the original text colors.
    SetConsoleTextAttribute(hStdout, wOldColorAttrs);
}

// The NewLine function handles carriage returns when the processed
// input mode is disabled. It gets the current cursor position
// and resets it to the first cell of the next row.
void NewLine(void)
{
    if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"), TEXT("Console
Error"), MB_OK);
        return;
    }

    csbiInfo.dwCursorPosition.X = 0;

    // If it is the last line in the screen buffer, scroll the buffer up
    if ((csbiInfo.dwSize.Y-1) == csbiInfo.dwCursorPosition.Y)
    {
        ScrollScreenBuffer(hStdout, 1);
    }

    // Otherwise, advance the cursor to the next line
    else csbiInfo.dwCursorPosition.Y += 1;

    if (! SetConsoleCursorPosition(hStdout, csbiInfo.dwCursorPosition))
    {
        MessageBox(NULL, TEXT("SetConsoleCursorPosition"), TEXT("Console
Error"), MB_OK);
        return;
    }
}
```

```
void ScrollScreenBuffer(HANDLE h, INT x)
{
    SMALL_RECT srctScrollRect, srctClipRect;
    CHAR_INFO chiFill;
    COORD coordDest;

    srctScrollRect.Left = 2;
    srctScrollRect.Top = 2;
    srctScrollRect.Right = csbiInfo.dwSize.X - x;
    srctScrollRect.Bottom = csbiInfo.dwSize.Y - x;

    // The destination for the scroll rectangle is one row up.
    coordDest.X = 0;
    coordDest.Y = 0;

    // The clipping rectangle is the same as the scrolling rectangle.
    // The destination row is left unchanged.
    srctClipRect = srctScrollRect;

    // Set the fill character and attributes.
    chiFill.Attributes = FOREGROUND_RED|FOREGROUND_INTENSITY;
    chiFill.Char.AsciiChar = (char)' ';

    // Scroll up one line.
    ScrollConsoleScreenBuffer(
        h,                 // screen buffer handle
        &srctScrollRect, // scrolling rectangle
        &srctClipRect,   // clipping rectangle
        coordDest,       // top left destination cell
        &chiFill);       // fill character and color
}
```

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



```
Type a line of text and press Enter, or q to quit: A line of text
Type a line of text and press Enter, or q to quit: Another line of text
Type a line of text and press Enter, or q to quit: more text
Type a line of text and press Enter, or q to quit: another line
Type a line of text and press Enter, or q to quit: q

Type q to quit: q
Press any key to continue . . .
```

## Reading and Writing Blocks of Characters and Attributes Example

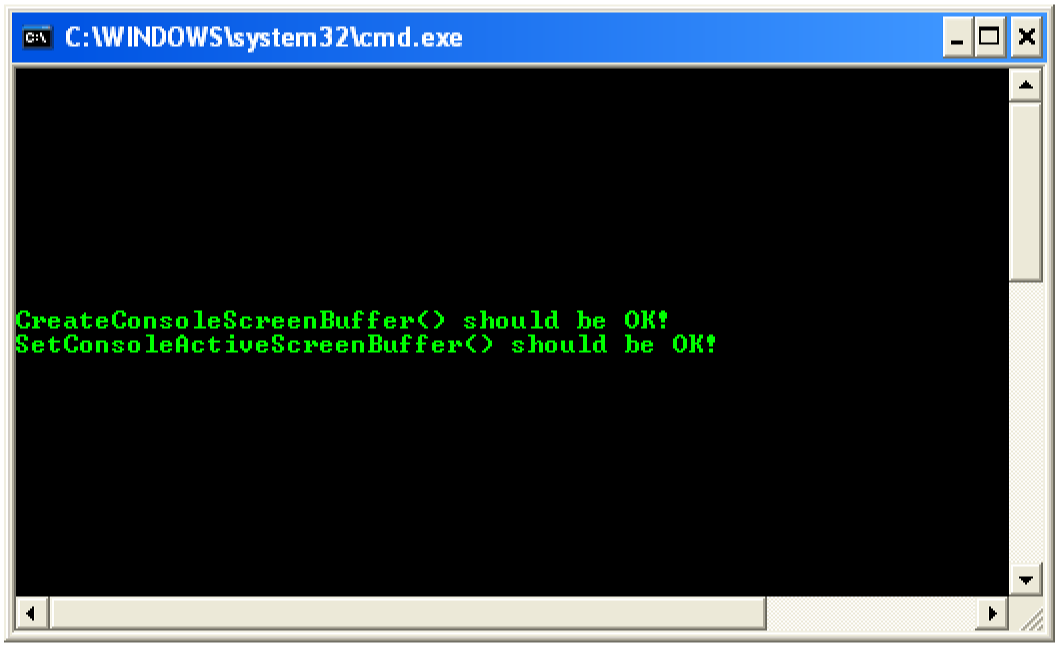
The ReadConsoleOutput() function copies a rectangular block of character and color attribute data from a console screen buffer into a destination buffer. The function treats the destination buffer as a two-dimensional array of CHAR_INFO structures. Similarly, the

WriteConsoleOutput() function copies a rectangular block of character and color attribute data from a source buffer to a console screen buffer.
The following example uses the CreateConsoleScreenBuffer() function to create a new screen buffer. After the SetConsoleActiveScreenBuffer() function makes this the active screen buffer, a block of characters and color attributes is copied from the top two rows of the STDOUT screen buffer into a temporary buffer. The data is then copied from the temporary buffer into the new active screen buffer. When the application is finished using the new screen buffer, it calls SetConsoleActiveScreenBuffer() to restore the original STDOUT screen buffer.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

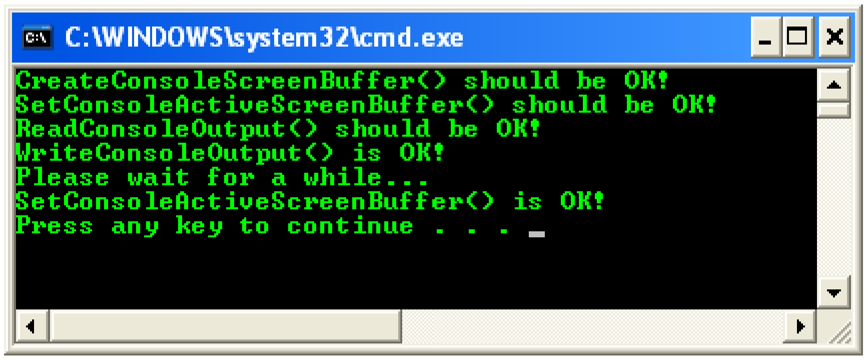
```
#include <windows.h>
#include <stdio.h>

int wmain(int argc, WCHAR **argv)
{
    HANDLE hStdout, hNewScreenBuffer;
    SMALL_RECT srctReadRect;
    SMALL_RECT srctWriteRect;
    CHAR_INFO chiBuffer[160]; // [2][80];
    COORD coordBufSize;
    COORD coordBufCoord;
    BOOL fSuccess;

    // Get a handle to the STDOUT screen buffer to copy from and
    // create a new screen buffer to copy to.
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    hNewScreenBuffer = CreateConsoleScreenBuffer(
       GENERIC_READ | GENERIC_WRITE,        // read/write access
       0,                          // not shared
       NULL,                       // default security attributes
       CONSOLE_TEXTMODE_BUFFER, // must be TEXTMODE
       NULL);                      // reserved; must be NULL

    if (hStdout == INVALID_HANDLE_VALUE || hNewScreenBuffer ==
INVALID_HANDLE_VALUE)
    {
        wprintf(L"CreateConsoleScreenBuffer() failed, error %d\n",
GetLastError());
        return 1;
    }
      else
            wprintf(L"CreateConsoleScreenBuffer() should be OK!\n");

    // Make the new screen buffer the active screen buffer.
    if (! SetConsoleActiveScreenBuffer(hNewScreenBuffer) )
    {
        wprintf(L"SetConsoleActiveScreenBuffer() failed, error %d\n",
GetLastError());
        return 1;
    }
      else
            wprintf(L"SetConsoleActiveScreenBuffer() should be OK!\n");

    // Set the source rectangle.
    srctReadRect.Top = 0;    // top left: row 0, col 0
    srctReadRect.Left = 0;
    srctReadRect.Bottom = 1; // bottom right: row 1, col 79
    srctReadRect.Right = 79;

    // The temporary buffer size is 2 rows x 80 columns.
    coordBufSize.Y = 2;
    coordBufSize.X = 80;

    // The top left destination cell of the temporary buffer is row 0, col 0.
    coordBufCoord.X = 0;
    coordBufCoord.Y = 0;

    // Copy the block from the screen buffer to the temp. buffer.
    fSuccess = ReadConsoleOutput(
       hStdout,        // screen buffer to read from
       chiBuffer,      // buffer to copy into
       coordBufSize,   // col-row size of chiBuffer
       coordBufCoord,  // top left dest. cell in chiBuffer
       &srctReadRect); // screen buffer source rectangle
    if (! fSuccess)
    {
       wprintf(L"ReadConsoleOutput() failed, error %d\n", GetLastError());
        return 1;
    }
      else
            wprintf(L"ReadConsoleOutput() should be OK!\n");

    // Set the destination rectangle.
    srctWriteRect.Top = 10;    // top lt: row 10, col 0
    srctWriteRect.Left = 0;
    srctWriteRect.Bottom = 11; // bottom rt: row 11, col 79
    srctWriteRect.Right = 79;

    // Copy from the temporary buffer to the new screen buffer.
    fSuccess = WriteConsoleOutput(
        hNewScreenBuffer, // screen buffer to write to
        chiBuffer,        // buffer to copy from
        coordBufSize,     // col-row size of chiBuffer
        coordBufCoord,    // top left src cell in chiBuffer
        &srctWriteRect);  // dest. screen buffer rectangle
    if (! fSuccess)
    {
        wprintf(L"WriteConsoleOutput() failed, error %d\n", GetLastError());
        return 1;
    }
      else
```

```
        wprintf(L"WriteConsoleOutput() is OK!\n");

       wprintf(L"Please wait for a while...\n");
    Sleep(5000);

    // Restore the original active screen buffer.
    if (!SetConsoleActiveScreenBuffer(hStdout))
    {
        wprintf(L"SetConsoleActiveScreenBuffer() failed, error %d\n",
GetLastError());
        return 1;
    }
      else
            wprintf(L"SetConsoleActiveScreenBuffer() is OK!\n");
}
```

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



```
CreateConsoleScreenBuffer() should be OK!
SetConsoleActiveScreenBuffer() should be OK!
```



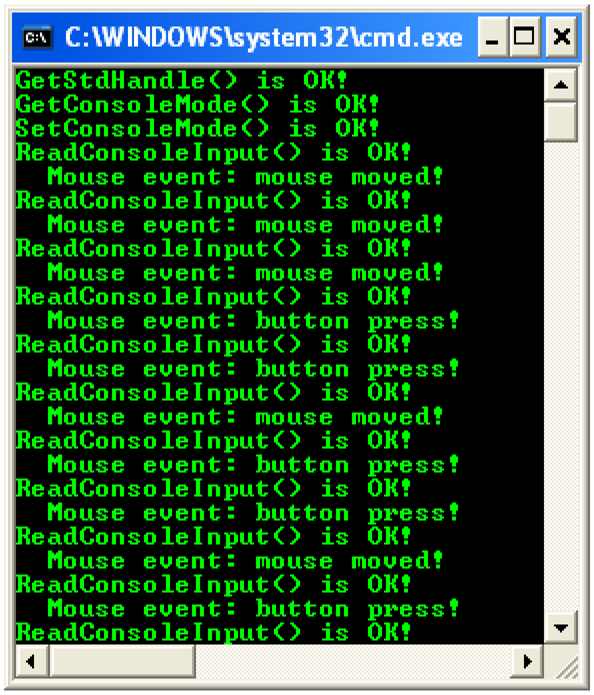
```
CreateConsoleScreenBuffer() should be OK!
SetConsoleActiveScreenBuffer() should be OK!
ReadConsoleOutput() should be OK!
WriteConsoleOutput() is OK!
Please wait for a while...
SetConsoleActiveScreenBuffer() is OK!
Press any key to continue . . .
```

**Reading Input Buffer Events Example**

The ReadConsoleInput() function can be used to directly access a console's input buffer. When a console is created, mouse input is enabled and window input is disabled. To ensure that the process receives all types of events, this example uses the SetConsoleMode() function to enable window and mouse input. Then it goes into a loop that reads and handles 100 console input events. For example, the message "Keyboard event" is displayed when the user presses a key and the message "Mouse event" is displayed when the user interacts with the mouse.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

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

// Prototypes
void ErrorExit(LPSTR);
void KeyEventProc(KEY_EVENT_RECORD);
void MouseEventProc(MOUSE_EVENT_RECORD);
void ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD);

int wmain(int argc, WCHAR **argv)
{
    HANDLE hStdin;
    DWORD cNumRead, fdwMode, fdwSaveOldMode, i;
    INPUT_RECORD irInBuf[128];
    int counter=0;

    // Get the standard input handle.
    hStdin = GetStdHandle(STD_INPUT_HANDLE);
    if (hStdin == INVALID_HANDLE_VALUE)
        ErrorExit("GetStdHandle()");
      else
            wprintf(L"GetStdHandle() is OK!\n");

    // Save the current input mode, to be restored on exit.
    if (!GetConsoleMode(hStdin, &fdwSaveOldMode) )
        ErrorExit("GetConsoleMode()");
      else
            wprintf(L"GetConsoleMode() is OK!\n");

    // Enable the window and mouse input events.
    fdwMode = ENABLE_WINDOW_INPUT | ENABLE_MOUSE_INPUT;
    if (!SetConsoleMode(hStdin, fdwMode))
        ErrorExit("SetConsoleMode()");
      else
            wprintf(L"SetConsoleMode() is OK!\n");

    // Loop to read and handle the input events.
    while (counter++ <= 30)
    {
        // Wait for the events
        if (!ReadConsoleInput(
                hStdin,      // input buffer handle
                irInBuf,     // buffer to read into
                128,         // size of read buffer
                &cNumRead) ) // number of records read
            ErrorExit("ReadConsoleInput()");
            else
                 wprintf(L"ReadConsoleInput() is OK!\n");

        // Dispatch the events to the appropriate handler
        for (i = 0; i < cNumRead; i++)
        {
            switch(irInBuf[i].EventType)
            {
                case KEY_EVENT: // keyboard input
                    KeyEventProc(irInBuf[i].Event.KeyEvent);
                    break;

                case MOUSE_EVENT: // mouse input
                    MouseEventProc(irInBuf[i].Event.MouseEvent);
                    break;

                case WINDOW_BUFFER_SIZE_EVENT: // scrn buf. resizing
                    ResizeEventProc(irInBuf[i].Event.WindowBufferSizeEvent);
                    break;

                case FOCUS_EVENT:  // disregard focus events

                case MENU_EVENT:   // disregard menu events
                    break;

                default:
                    ErrorExit("Unknown event type");
                    break;
            }
        }
    }
    return 0;
}

void ErrorExit (LPSTR lpszMessage)
{
    fprintf(stderr, "%s\n", lpszMessage);
    ExitProcess(0);
}

void KeyEventProc(KEY_EVENT_RECORD ker)
{
    wprintf(L"  Key event: ");

    if(ker.bKeyDown)
        wprintf(L"key pressed!\n");
    else
            wprintf(L"key released!\n");
}

void MouseEventProc(MOUSE_EVENT_RECORD mer)
{
    wprintf(L"  Mouse event: ");

    switch(mer.dwEventFlags)
    {
        case 0:
            wprintf(L"button press!\n");
            break;
        case DOUBLE_CLICK:
            wprintf(L"double click!\n");
            break;
        case MOUSE_HWHEELED:
            wprintf(L"horizontal mouse wheel!\n");
            break;
        case MOUSE_MOVED:
            wprintf(L"mouse moved!\n");
            break;
        case MOUSE_WHEELED:
            wprintf(L"vertical mouse wheel!\n");
            break;
        default:
            wprintf(L"unknown\n");
            break;
    }
}

VOID ResizeEventProc(WINDOW_BUFFER_SIZE_RECORD wbsr)
{
    wprintf(L"Resize event!\n");
}
```

Build and run the project. Click, move and double click the mouse. The following screenshot is a sample output.



```
GetStdHandle() is OK!
GetConsoleMode() is OK!
SetConsoleMode() is OK!
ReadConsoleInput() is OK!
  Mouse event: mouse moved!
ReadConsoleInput() is OK!
  Mouse event: mouse moved!
ReadConsoleInput() is OK!
  Mouse event: mouse moved!
ReadConsoleInput() is OK!
  Mouse event: button press!
ReadConsoleInput() is OK!
  Mouse event: button press!
ReadConsoleInput() is OK!
  Mouse event: mouse moved!
ReadConsoleInput() is OK!
  Mouse event: button press!
ReadConsoleInput() is OK!
  Mouse event: button press!
ReadConsoleInput() is OK!
  Mouse event: mouse moved!
ReadConsoleInput() is OK!
  Mouse event: button press!
ReadConsoleInput() is OK!
```

### Clearing the Console Screen

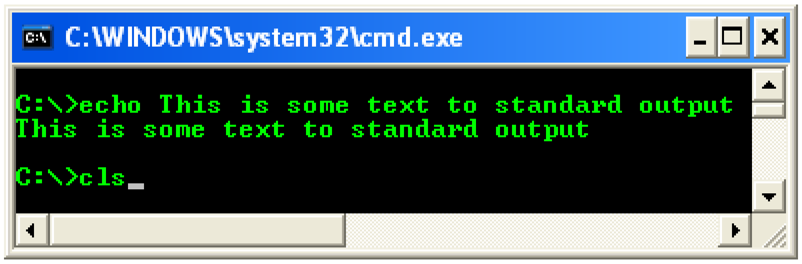
There are two ways to clear the screen in a console application.

### Clear Screen: Example 1

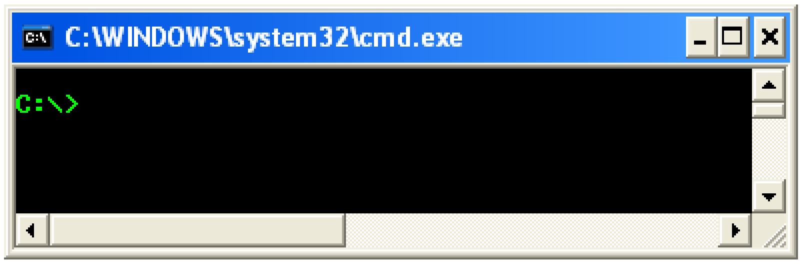
The first method is to use the C run-time system function. The system function invokes the cls command provided by the command interpreter to clear the screen.

```
#include <stdlib.h>

int wmain(int argc, WCHAR **argv)
{
   system("cls");
   return 0;
}
```

It is similar to running the **cls** command at the command prompt.



```
C:\>echo This is some text to standard output
This is some text to standard output

C:\>cls_
```



```
C:\>
```

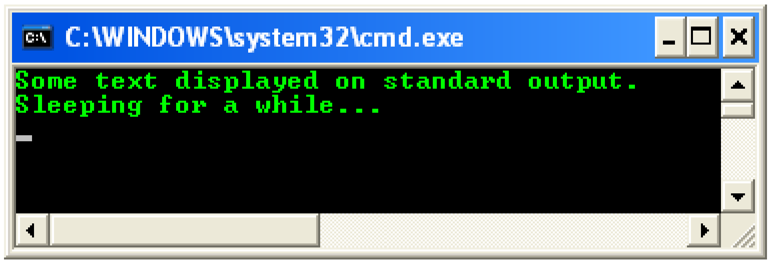
**Clear Screen: Example 2**

The second method is to write a function to programmatically clear the screen using the FillConsoleOutputCharacter() and FillConsoleOutputAttribute() functions. The following sample code demonstrates this technique.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

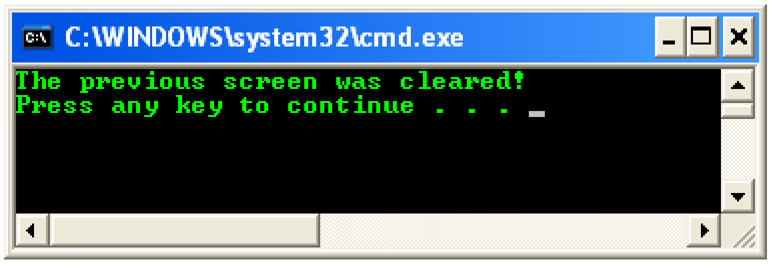
```
#include <windows.h>
#include <stdio.h>

void cls(HANDLE hConsole)
{
	// home for the cursor
	COORD coordScreen = {0, 0};
	DWORD cCharsWritten;
	CONSOLE_SCREEN_BUFFER_INFO csbi;
	DWORD dwConSize;

	// Get the number of character cells in the current buffer
	if(!GetConsoleScreenBufferInfo(hConsole, &csbi))
		return;
	dwConSize = csbi.dwSize.X * csbi.dwSize.Y;

	// Fill the entire screen with blanks
	if(!FillConsoleOutputCharacter(hConsole, (TCHAR)' ', dwConSize,
coordScreen, &cCharsWritten))
		return;
```

```
      // Get the current text attribute.
      if(!GetConsoleScreenBufferInfo(hConsole, &csbi))
            return;

      // Set the buffer's attributes accordingly.
      if(!FillConsoleOutputAttribute(hConsole, csbi.wAttributes, dwConSize,
coordScreen, &cCharsWritten))
            return;

      // Put the cursor at its home coordinates.
      SetConsoleCursorPosition(hConsole, coordScreen);
      wprintf(L"The previous screen was cleared!\n");
}

int wmain(int argc, WCHAR **argv)
{
    HANDLE hStdout;

      wprintf(L"Some text displayed on standard output.\n");
      wprintf(L"Sleeping for a while...\n");
      // sleep for 5000 ms
      Sleep(5000);

      // Clear screen and exit
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    cls(hStdout);
      return 0;
}
```

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



```
Some text displayed on standard output.
Sleeping for a while...
_
```



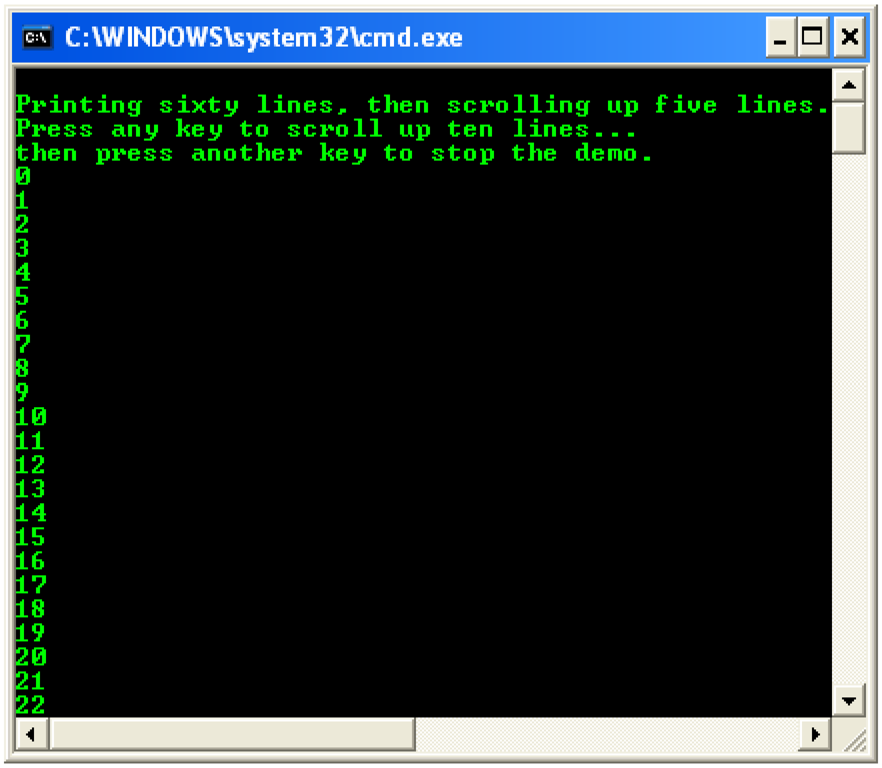
```
The previous screen was cleared!
Press any key to continue . . . _
```

## Scrolling a Screen Buffer's Window Example

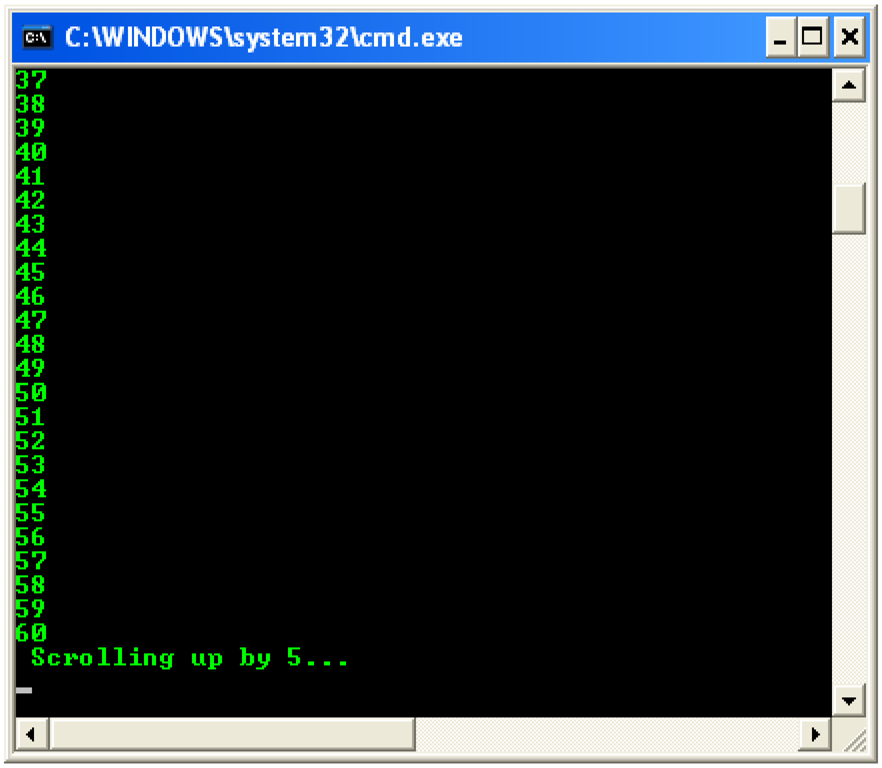
The SetConsoleWindowInfo() function can be used to scroll the contents of a screen buffer in the console window. This function can also change the window size. The function can either specify the new upper left and lower right corners of the console screen buffer's window as absolute screen buffer coordinates or specify the changes from the current window coordinates. The function fails if the specified window coordinates are outside the boundaries of the console screen buffer.
The following example scrolls the view of the console screen buffer up by modifying the window coordinates returned by the GetConsoleScreenBufferInfo() function. The ScrollByAbsoluteCoord() function demonstrates how to specify absolute coordinates, while the ScrollByRelativeCoord() function demonstrates how to specify relative coordinates.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>
// For _getch()
#include <conio.h>

// Global variable
HANDLE hStdout;

int ScrollByAbsoluteCoord(int iRows)
{
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    SMALL_RECT srctWindow;

    // Get the current screen buffer size and window position.
    if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        wprintf(L"GetConsoleScreenBufferInfo() failed, error %d\n",
GetLastError());
        return 1;
    }

    // Set srctWindow to the current window size and location.
    srctWindow = csbiInfo.srWindow;

    // Check whether the window is too close to the screen buffer top
    if (srctWindow.Top >= iRows)
    {
        srctWindow.Top -= iRows;      // move top up
        srctWindow.Bottom -= iRows;   // move bottom up

        if (!SetConsoleWindowInfo(
                   hStdout,           // screen buffer handle
                   TRUE,              // absolute coordinates
                   &srctWindow))      // specifies new location
        {
            wprintf(L"SetConsoleWindowInfo() failed, error %d\n",
GetLastError());
            return 1;
        }
        return iRows;
    }
    else
    {
        wprintf(L"\nCannot scroll; the window is too close to the top.\n");
        return 1;
    }
}

int ScrollByRelativeCoord(int iRows)
{
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    SMALL_RECT srctWindow;

    // Get the current screen buffer window position.
    if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        wprintf(L"GetConsoleScreenBufferInfo(), error %d\n", GetLastError());
        return 1;
    }

    // Check whether the window is too close to the screen buffer top
    if (csbiInfo.srWindow.Top >= iRows)
    {
        srctWindow.Top = -iRows;      // move top up
        srctWindow.Bottom = -iRows;   // move bottom up
        srctWindow.Left = 0;          // no change
        srctWindow.Right = 0;         // no change

        if (!SetConsoleWindowInfo(
                   hStdout,           // screen buffer handle
                   FALSE,             // relative coordinates
                   &srctWindow))      // specifies new location
        {
            wprintf(L"SetConsoleWindowInfo(), error %d\n", GetLastError());
            return 0;
        }
        return iRows;
    }
    else
    {
        wprintf(L"\nCannot scroll; the window is too close to the top.\n");
        return 0;
    }
}

int wmain(int argc, WCHAR **argv)
{
    int i;
    wprintf(L"\nPrinting sixty lines, then scrolling up five lines.\n");
    wprintf(L"Press any key to scroll up ten lines...\n");
    wprintf(L"then press another key to stop the demo.\n");

    for(i=0; i<=60; i++)
        wprintf(L"%d\n", i);

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    if(ScrollByAbsoluteCoord(5))
      {
            wprintf(L" Scrolling up by 5...\n");
            _getch();
      }
    else
            return 1;

    if(ScrollByRelativeCoord(10))
      {
            wprintf(L" Scrolling up by 10...\n");
            _getch();
      }
    else
            return 1;

      return 0;
}
```

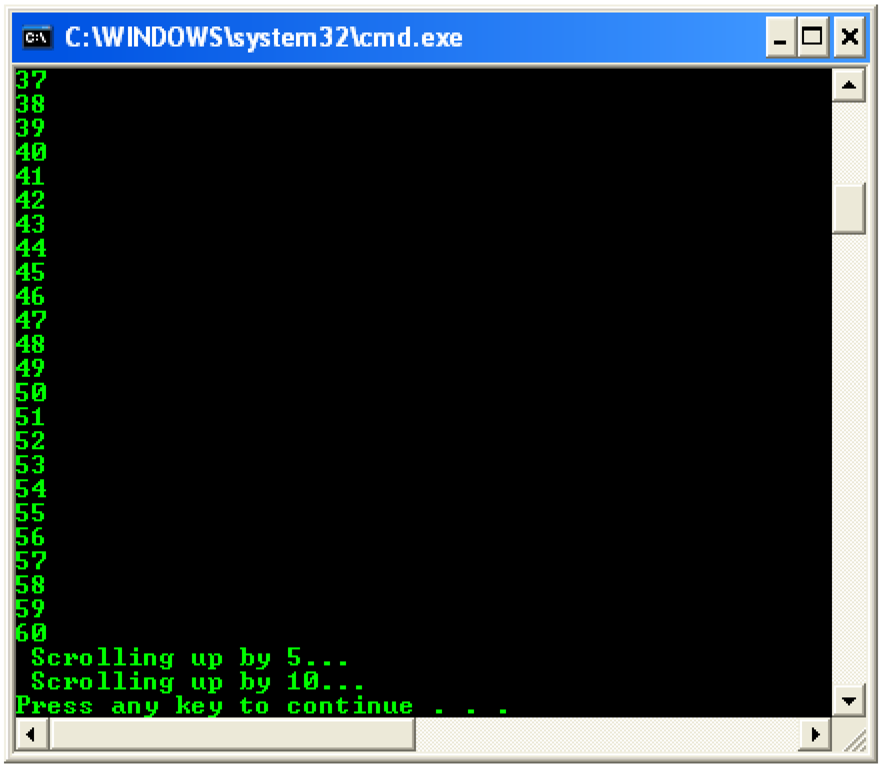
Build and run the project. The following screenshots are sample outputs. Notice the scroll bar movement.



```
Printing sixty lines, then scrolling up five lines.
Press any key to scroll up ten lines...
then press another key to stop the demo.
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```



```
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
 Scrolling up by 5...
_
```



```
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
 Scrolling up by 5...
 Scrolling up by 10...
Press any key to continue . . .
```

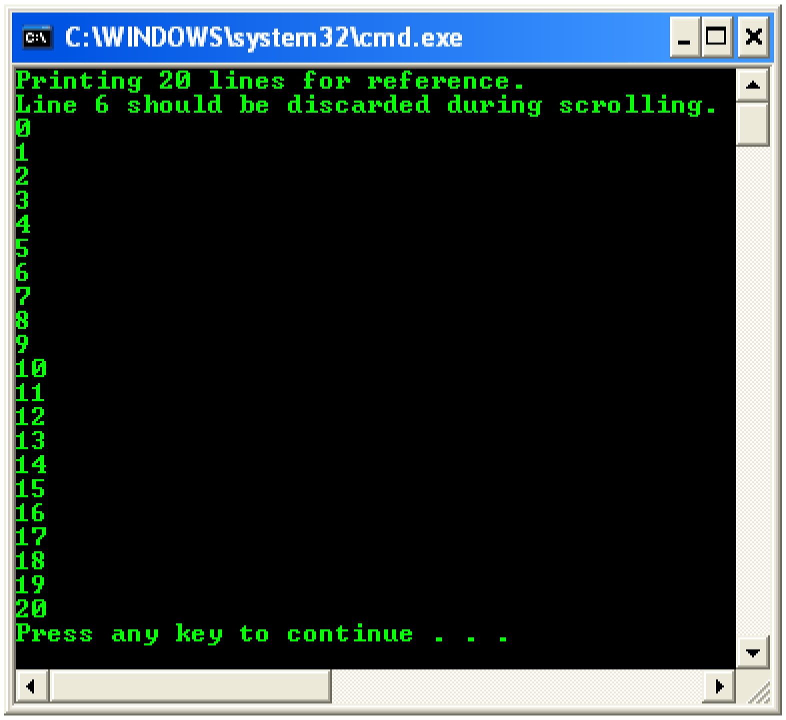
**Scrolling a Screen Buffer's Contents Example**

The ScrollConsoleScreenBuffer() function moves a block of character cells from one part of a screen buffer to another part of the same screen buffer. The function specifies the upper left and lower right cells of the source rectangle to be moved and the destination coordinates of the new location for the upper left cell. The character and color data in the source cells is moved to the new location, and any cells left empty by the move are filled in with a specified character and color. If a clipping rectangle is specified, the cells outside of it are left unchanged.
ScrollConsoleScreenBuffer() can be used to delete a line by specifying coordinates of the first cell in the line as the destination coordinates and specifying a scrolling rectangle that includes all the rows below the line.
The following example shows the use of a clipping rectangle to scroll only the bottom 15 rows of the console screen buffer. The rows in the specified rectangle are scrolled up one line at a time, and the top row of the block is discarded. The contents of the console screen buffer outside the clipping rectangle are left unchanged.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

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

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

int wmain(int argc, WCHAR **argv)
{
    HANDLE hStdout;
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    SMALL_RECT srctScrollRect, srctClipRect;
    CHAR_INFO chiFill;
    COORD coordDest;
    int i;

    wprintf(L"Printing 20 lines for reference.\n");
    wprintf(L"Line 6 should be discarded during scrolling.\n");

    for(i=0; i<=20; i++)
      {
        wprintf(L"%d\n", i);
            // Just for fun
            if(i == 4)
                  Sleep(1000);
      }

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);

    if(hStdout == INVALID_HANDLE_VALUE)
    {
        wprintf(L"GetStdHandle() failed with error %d\n", GetLastError());
        return 1;
    }

    // Get the screen buffer size.
    if(!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        wprintf(L"GetConsoleScreenBufferInfo() failed, error %d\n",
GetLastError());
        return 1;
    }

    // The scrolling rectangle is the bottom 15 rows of the screen buffer
    srctScrollRect.Top = csbiInfo.dwSize.Y - 16;
    srctScrollRect.Bottom = csbiInfo.dwSize.Y - 1;
    srctScrollRect.Left = 0;
    srctScrollRect.Right = csbiInfo.dwSize.X - 1;

    // The destination for the scroll rectangle is one row up
    coordDest.X = 0;
    coordDest.Y = csbiInfo.dwSize.Y - 17;

    // The clipping rectangle is the same as the scrolling rectangle.
    // The destination row is left unchanged
    srctClipRect = srctScrollRect;

    // Fill the bottom row with green blanks
    chiFill.Attributes = BACKGROUND_GREEN | FOREGROUND_RED;
    chiFill.Char.AsciiChar = (char)' ';

    // Scroll up one line
    if(!ScrollConsoleScreenBuffer(
        hStdout,         // screen buffer handle
        &srctScrollRect, // scrolling rectangle
        &srctClipRect,   // clipping rectangle
        coordDest,       // top left destination cell
        &chiFill))       // fill character and color
    {
        wprintf(L"ScrollConsoleScreenBuffer() failed, error %d\n",
GetLastError());
        return 1;
    }
}
```

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



```
Printing 20 lines for reference.
Line 6 should be discarded during scrolling.
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Press any key to continue . . .
```

**Registering a Control Handler Function Example**

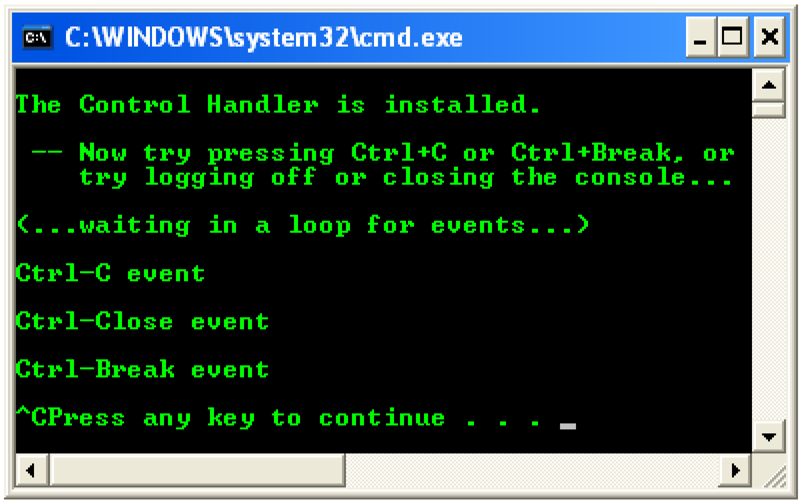
This is an example of the SetConsoleCtrlHandler() function that is used to install a control handler.
When a CTRL+C signal is received, the control handler returns TRUE, indicating that it has handled the signal. Doing this prevents other control handlers from being called.
When a CTRL_CLOSE_EVENT signal is received, the control handler returns TRUE, causing the system to display a dialog box that gives the user the choice of terminating the process and

closing the console or allowing the process to continue execution. If the user chooses not to terminate the process, the system closes the console when the process finally terminates.
When a CTRL+BREAK, CTRL_LOGOFF_EVENT, or CTRL_SHUTDOWN_EVENT signal is received, the control handler returns FALSE. Doing this causes the signal to be passed to the next control handler function. If no other control handlers have been registered or none of the registered handlers returns TRUE, the default handler will be used, resulting in the process being terminated.
Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

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

BOOL CtrlHandler(DWORD fdwCtrlType)
{
  switch(fdwCtrlType)
  {
        // Note: Console functions, or any C run-time functions that
        // call console functions, may not work reliably during processing
        // of any of the three signals mentioned previously.
        // The reason is that some or all of the internal console cleanup
        // routines may have been called before executing the process signal
handler.
        // Handle the CTRL-C signal.
    case CTRL_C_EVENT:
      printf("Ctrl-C event\n\n");
      Beep(750, 300);
      return(TRUE);

    // CTRL-CLOSE: confirm that the user wants to exit
    case CTRL_CLOSE_EVENT:
      Beep(600, 200);
      wprintf(L"Ctrl-Close event\n\n");
      return(TRUE);

    // Pass other signals to the next handler
    case CTRL_BREAK_EVENT:
      Beep(900, 200);
      wprintf(L"Ctrl-Break event\n\n");
      return FALSE;

    case CTRL_LOGOFF_EVENT:
      Beep(1000, 200);
      wprintf(L"Ctrl-Logoff event\n\n");
      return FALSE;

    case CTRL_SHUTDOWN_EVENT:
      Beep(750, 500);
      wprintf(L"Ctrl-Shutdown event\n\n");
```

```
        return FALSE;

      default:
        return FALSE;
    }
}

int wmain(int argc, WCHAR **argv)
{
  if(SetConsoleCtrlHandler((PHANDLER_ROUTINE) CtrlHandler, TRUE))
  {
    wprintf(L"\nThe Control Handler is installed.\n" );
    wprintf(L"\n -- Now try pressing Ctrl+C or Ctrl+Break, or" );
    wprintf(L"\n    try logging off or closing the console...\n" );
    wprintf(L"\n(...waiting in a loop for events...)\n\n" );

      // While true
    while(1){ }
  }
  else
    wprintf(L"\nERROR: Could not set control handler");

  return 0;
}
```

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



```
The Control Handler is installed.

 -- Now try pressing Ctrl+C or Ctrl+Break, or
    try logging off or closing the console...

(...waiting in a loop for events...)

Ctrl-C event

Ctrl-Close event

Ctrl-Break event

^CPress any key to continue . . .
```