控制台窗口界面的编程控制（一）
2008-05-23 18:49
2002-09-13 09:31 作者： 丁有和 出处： yesky 责任编辑：

文本界面的控制台应用程序开发是深入学习 C++、掌握交互系统的实现方法的最简单的一种手段。然而，Visual C++的 C++专用库却没有 TC 所支持的文本(字符)屏幕控制函数，为此本系列文章从一般控制步骤、控制台窗口操作、文本(字符)控制、滚动和移动、光标、键盘和鼠标等几个方面讨论控制台窗口界面的编程控制方法。

在众多 C++开发工具中，由于 Microsoft 本身的独特优势，选用 Visual C++已越来越被众多学习者所接受。显然，现今如果还再把 TC 作为开发环境的话，不仅没有必要，而且也不利于向 Windows 应用程序开发的过渡。然而，Visual C++的 C++专用库却没有 TC 所支持的文本屏幕(控制台窗口)控制函数(相应的头文件是 conio.h)。这必然给 C++学习者在文本界面设计和编程上带来诸多不便。要知道，文本界面设计是一种深入学习 C++、掌握交互系统的实现方法的最简单的一种手段，它不像 C++的 Windows 图形界面应用程序，涉及知识过多。为此，本系列文章来讨论在 Visual C++ 6.0 开发环境中，如何编写具有美观清晰的控制台窗口界面的 C++应用程序。

一、概述

所谓控制台应用程序，就是指那些需要与传统 DOS 操作系统保持某种程序的兼容，同时又不需要为用户提供完善界面的程序。简单地讲，就是指在 Windows 环境下运行的 DOS 程序。一旦 C++控制台应用程序在 Windows 9x/NT/2000 操作系统中运行后，就会弹出一个窗口。例如下列过程：

单击 Visual C++标准工具栏上的“New Text File”按钮，打开一个新的文档窗口。

选择 File | Save 菜单或按快捷键 Ctrl+S 或单击标准工具栏的 Save 按钮，弹出“保存为”文件对话框。将文件名为“Hello.cpp” (注意扩展名.cpp 不能省略)。

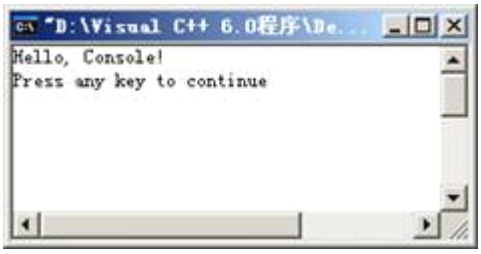
在文档窗口中输入下列代码：

```
#include
void main()
{
cout<<"Hello, Console!"< }
```

单击小型编译工具栏中的“Build”按钮或按 F7 键，系统出现一个对话框，询问是否将此项目的工作文件夹设定源文件所在的文件夹，单击[是]按钮，系统开始编译。

单击小型编译工具栏中的“Execute Program”按钮或按 Ctrl+F5 键，运行刚才的程序。

程序运行后，弹出下图的窗口。



这就是控制台窗口，与传统的 DOS 屏幕窗口相比最主要的区别有：

(1) 默认的控制台窗口有系统菜单和标题，它是一个内存缓冲区窗口，缓冲区大小取决于 Windows 操作系统的分配；而 DOS 屏幕是一种物理窗口，不具有 Windows 窗口特性，其大小取决于 ROM BIOS 分配的内存空间。

(2) 控制台窗口的文本操作是调用低层的 Win32 APIs，而 DOS 屏幕的文本操作是通过调用 BIOS 的 16(10h)中断而实现的。

(3) 默认的控制台窗口可以接收键盘和鼠标的输入信息，设备驱动由 Windows 管理，而 DOS 屏幕窗口接收鼠标时需要调用 33h 中断，且鼠标设备驱动程序由自己安装。

二、控制台文本窗口的一般控制步骤

在 Visual C++ 6.0 中，控制台窗口界面的一般编程控制步骤如下：

调用 GetStdHandle 获取当前的标准输入(STDIN)和标准输出(STDOUT)设备句柄。函数原型为：

```
HANDLE GetStdHandle( DWORD nStdHandle );
```

其中，nStdHandle 可以是 STD_INPUT_HANDLE(标准输入设备句柄)、STD_OUTPU

T_HANDLE(标准输出设备句柄)和 STD_ERROR_HANDLE(标准错误句柄)。需要说明的是，“句柄”是 Windows 最常用的概念。它通常用来标识 Windows 资源(如菜单、图标、窗口等)和设备等对象。虽然可以把句柄理解为是一个指针变量类型，但它不是对象所在的地址指针，而是作为 Windows 系统内部表的索引值来使用的。

调用相关文本界面控制的 API 函数。这些函数可分为三类。一是用于控制台窗口操作的函数(包括窗口的缓冲区大小、窗口前景字符和背景颜色、窗口标题、大小和位置等)；二是用于控制台输入输出的函数(包括字符属性操作函数)；其他的函数并为最后一类。

调用 CloseHandle()来关闭输入输出句柄。

注意，在程序中还必须包含头文件 windows.h。下面看一个程序：

```
#include
#include
#include
void main()
{
HANDLE hOut;
hout = GetStdHandle(STD_OUTPUT_HANDLE);
// 获取标准输出设备句柄
CONSOLE_SCREEN_BUFFER_INFO bInfo; // 窗口信息
GetConsoleScreenBufferInfo(hOut, &bInfo );
// 获取窗口信息
printf("\n\nThe soul selects her own society,\n");
printf("Then shuts the door;\n");
printf("On her devine majority\n");
printf("Obtrude no more.\n\n");
_getch();
COORD pos = {0, 0};
FillConsoleOutputCharacter(hOut, ' ', bInfo.dwSize.X *
bInfo.dwSize.Y, pos, NULL);
// 向窗口中填充字符以获得清屏的效果
CloseHandle(hOut); // 关闭标准输出设备句柄
}
```

程序中，COORD 和 CONSOLE_SCREEN_BUFFER_ INFO 是 wincon.h 定义的控制台结构体类型，其原型如下：

```
// 坐标结构体
typedef struct _COORD {
SHORT X;
SHORT Y;
} COORD;

// 控制台窗口信息结构体
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
COORD dwSize; // 缓冲区大小
COORD dwCursorPosition; // 当前光标位置
WORD wAttributes; // 字符属性
SMALL_RECT srWindow; // 当前窗口显示的大小和位置
COORD dwMaximumWindowSize; // 最大的窗口缓冲区大小
} CONSOLE_SCREEN_BUFFER_INFO ;
```

还需要说明的是，虽然在C++中，iostream.h定义了cin和cout的标准输入和输出流对象。但它们只能实现基本的输入输出操作，对于控制台窗口界面的控制却无能为力，而且不能与stdio.h和conio.h友好相处，因为iostream.h和它们是C++两套不同的输入输出操作方式，使用时要特别注意。

三、控制台窗口操作

用于控制台窗口操作的API函数如下：

GetConsoleScreenBufferInfo 获取控制台窗口信息
GetConsoleTitle 获取控制台窗口标题
ScrollConsoleScreenBuffer 在缓冲区中移动数据块
SetConsoleScreenBufferSize 更改指定缓冲区大小
SetConsoleTitle 设置控制台窗口标题

SetConsoleWindowInfo 设置控制台窗口信息

此外，还有窗口字体、显示模式等控制函数，这里不再细说。下列举一个示例，程序如下：

```
#include
#include
#include
void main()
{
HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
// 获取标准输出设备句柄
CONSOLE_SCREEN_BUFFER_INFO bInfo; // 窗口缓冲区信息
GetConsoleScreenBufferInfo(hOut, bInfo );
// 获取窗口缓冲区信息
char strTitle[255];
GetConsoleTitle(strTitle, 255); // 获取窗口标题
printf("当前窗口标题是：%s\n", strTitle);
_getch();
SetConsoleTitle("控制台窗口操作"); // 获取窗口标题
_getch();
COORD size = {80, 25};
SetConsoleScreenBufferSize(hOut,size); // 重新设置缓冲区大小
_getch();
SMALL_RECT rc = {0,0, 80-1, 25-1}; // 重置窗口位置和大小
SetConsoleWindowInfo(hOut,true ,&rc);
CloseHandle(hOut); // 关闭标准输出设备句柄
}
```

需要说明的是，控制台窗口的原点坐标是(0, 0)，而最大的坐标是缓冲区大小减 1，例如当缓冲区大小为 80*25 时，其最大的坐标是(79, 24)。

四、文本属性操作

与 DOS 字符相似，控制台窗口中的字符也有相应的属性。这些属性分为：文本的前景色、背景色和双字节字符集(DBCS)属性三种。事实上，我们最关心是文本颜色，这样可以构造出美观的界面。颜色属性都是一些预定义标识：

FOREGROUND_BLUE 蓝色

FOREGROUND_GREEN 绿色
FOREGROUND_RED 红色
FOREGROUND_INTENSITY 加强
BACKGROUND_BLUE 蓝色背景
BACKGROUND_GREEN 绿色背景
BACKGROUND_RED 红色背景
BACKGROUND_INTENSITY 背景色加强
COMMON_LVB_REVERSE_VIDEO 反色

与文本属性相关的主要函数有：

```
BOOL FillConsoleOutputAttribute( // 填充字符属性
HANDLE hConsoleOutput, // 句柄
WORD wAttribute, // 文本属性
DWORD nLength, // 个数
COORD dwWriteCoord, // 开始位置
LPDWORD lpNumberOfAttrsWritten // 返回填充的个数
);

BOOL SetConsoleTextAttribute( // 设置 WriteConsole 等函数的字符属
性
HANDLE hConsoleOutput, // 句柄
WORD wAttributes // 文本属性
);

BOOL WriteConsoleOutputAttribute( // 在指定位置处写属性
HANDLE hConsoleOutput, // 句柄
CONST WORD *lpAttribute, // 属性
DWORD nLength, // 个数
COORD dwWriteCoord, // 起始位置
LPDWORD lpNumberOfAttrsWritten // 已写个数
);
```

另外，获取当前控制台窗口的文本属性是通过调用函数 GetConsoleScreenBufferInfo 后，在 CONSOLE_SCREEN_ BUFFER_INFO 结构成员 wAttributes 中得到。

五、文本输出

文本输出函数有：

```
BOOL FillConsoleOutputCharacter( // 填充指定数据的字符
HANDLE hConsoleOutput, // 句柄
TCHAR cCharacter, // 字符
DWORD nLength, // 字符个数
COORD dwWriteCoord, // 起始位置
LPDWORD lpNumberOfCharsWritten // 已写个数
);

BOOL WriteConsole( // 在当前光标位置处插入指定数量的字符
HANDLE hConsoleOutput, // 句柄
CONST VOID *lpBuffer, // 字符串
DWORD nNumberOfCharsToWrite, // 字符个数
LPDWORD lpNumberOfCharsWritten, // 已写个数
LPVOID lpReserved // 保留
);

BOOL WriteConsoleOutput( // 向指定区域写带属性的字符
HANDLE hConsoleOutput, // 句柄
CONST CHAR_INFO *lpBuffer, // 字符数据区
COORD dwBufferSize, // 数据区大小
COORD dwBufferCoord, // 起始坐标
PSMALL_RECT lpWriteRegion // 要写的区域
);

BOOL WriteConsoleOutputCharacter( // 在指定位置处插入指定数量的
字符
HANDLE hConsoleOutput, // 句柄
LPCTSTR lpCharacter, // 字符串
DWORD nLength, // 字符个数
COORD dwWriteCoord, // 起始位置
LPDWORD lpNumberOfCharsWritten // 已写个数
);
```

可以看出：WriteConsoleOutput 函数功能相当于 SetConsoleTextAttribute 和 WriteConsole 的功能。而 WriteConsoleOutputCharacter 函数相当于 SetConsoleCursorPosition(设置光标位置)和 WriteConsole 的功能。不过在具体使用要注意它们的区别。

六、文本操作示例

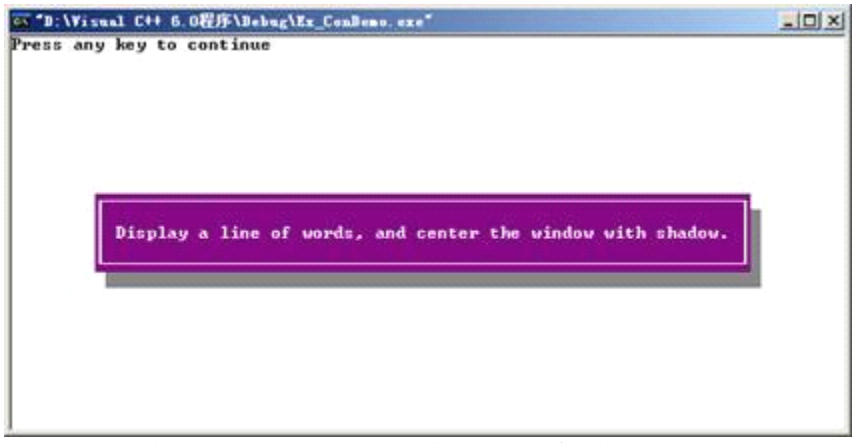
下面看一个示例程序：

```
#include <windows.h>
HANDLE hOut;
void ShadowWindowLine(char *str); // 在具有阴影效果的窗口中显示一行字符，窗口为居中显示
void DrawBox(bool bSingle, SMALL_RECT rc); // 绘制边框
void main()
{
  hOut = GetStdHandle(STD_OUTPUT_HANDLE); // 获取标准输出设备句柄
  SetConsoleOutputCP(437); // 设置代码页
  ShadowWindowLine("Display a line of words, and center the window with shadow.");
  CloseHandle(hOut); // 关闭标准输出设备句柄
}

void ShadowWindowLine(char *str)
{
  CONSOLE_SCREEN_BUFFER_INFO bInfo; // 窗口缓冲区信息
  GetConsoleScreenBufferInfo( hOut, &bInfo ); // 获取窗口缓冲区信息
  // 计算显示窗口大小和位置
  int x1, y1, x2, y2, chNum = strlen(str);
  x1 = (bInfo.dwSize.X - chNum)/2 - 2;
  y1 = bInfo.dwSize.Y/2 - 2;
  x2 = x1 + chNum + 4;
  y2 = y1 + 5;
  WORD att1 = BACKGROUND_INTENSITY; // 阴影属性
  WORD att0 = FOREGROUND_RED |FOREGROUND_GREEN |FOREGROUND_BLUE |
              FOREGROUND_INTENSITY |
              BACKGROUND_RED | BACKGROUND_BLUE; // 文本属性

  WORD attText = FOREGROUND_RED |FOREGROUND_INTENSITY; // 文本属性
  // 设置阴影
  COORD posShadow = {x1+1, y1+1}, posText = {x1, y1};
  for (int i=0; i<5; i++){
```

```
    FillConsoleOutputAttribute(hOut, att1, chNum + 4, posShadow,
NULL);
    posShadow.Y++;
  }
  // 填充窗口背景
  for (i=0; i<5; i++){
    FillConsoleOutputAttribute(hOut, att0, chNum + 4, posText,
NULL);
    posText.Y++;
  }
  // 写文本和边框
  posText.X = x1 + 2;
  posText.Y = y1 + 2;
  WriteConsoleOutputCharacter(hOut, str, strlen(str), posText,
NULL);
  SMALL_RECT rc = {x1, y1, x2-1, y2-1};
  DrawBox(true, rc);
  SetConsoleTextAttribute(hOut, bInfo.wAttributes); // 恢复原来
的属性
}

void DrawBox(bool bSingle, SMALL_RECT rc)
{
  char chBox[6];
  if (bSingle) {
    chBox[0] = (char)0xda; // 左上角点
    chBox[1] = (char)0xbf; // 右上角点
    chBox[2] = (char)0xc0; // 左下角点
    chBox[3] = (char)0xd9; // 右下角点
    chBox[4] = (char)0xc4; // 水平
    chBox[5] = (char)0xb3; // 坚直
  } else {
    chBox[0] = (char)0xc9; // 左上角点
    chBox[1] = (char)0xbb; // 右上角点
    chBox[2] = (char)0xc8; // 左下角点
    chBox[3] = (char)0xbc; // 右下角点
    chBox[4] = (char)0xcd; // 水平
    chBox[5] = (char)0xba; // 坚直
  }
  COORD pos = {rc.Left, rc.Top};
```

```
  WriteConsoleOutputCharacter(hOut, &chBox[0], 1, pos, NULL);

  for (pos.X = rc.Left + 1; pos.X
WriteConsoleOutputCharacter(hOut, &chBox[4], 1, pos, NULL);

  pos.X = rc.Right;
  WriteConsoleOutputCharacter(hOut, &chBox[1], 1, pos, NULL);

  for (pos.Y = rc.Top+1; pos.Y  {
    pos.X = rc.Left;
    WriteConsoleOutputCharacter(hOut, &chBox[5], 1, pos, NULL);
    pos.X = rc.Right;
    WriteConsoleOutputCharacter(hOut, &chBox[5], 1, pos, NULL);
  }
  pos.X = rc.Left; pos.Y = rc.Bottom;
  WriteConsoleOutputCharacter(hOut, &chBox[2], 1, pos, NULL);

  for (pos.X = rc.Left + 1; pos.X
WriteConsoleOutputCharacter(hOut, &chBox[4], 1, pos, NULL);

  pos.X = rc.Right;
  WriteConsoleOutputCharacter(hOut, &chBox[3], 1, pos, NULL);
}
```

程序运行结果如下图所示。



需要说明的是，上述程序在不同的字符代码页面(code page)下显示的结果是不同的。例如，中文 Windows 操作系统的默认代码页是简体中文(936)，在该代码页面下值超过 128 的单字符在 Windows NT/XP 是显示不出来的。下表列出了可以使用的代码页。

代码页(Code page)	说 明
1258	越南文
1257	波罗的海文
1256	阿拉伯文
1255	希伯来文
1254	土耳其语
1253	希腊文
1252	拉丁文(ANSI)
1251	斯拉夫文
1250	中欧文
950	繁体中文
949	韩文
936	简体中文
932	日文
874	泰文
850	使用多种语言(MS-DOS 拉丁文)
437	MS-DOS 美语/英语

七、滚动和移动

ScrollConsoleScreenBuffer 是实现文本区滚动和移动的 API 函数。它可以将指定的一块文本区域移动到另一个区域，被移空的那块区域由指定字符填充。函数的原型如下：

```
BOOL ScrollConsoleScreenBuffer(
    HANDLE hConsoleOutput, // 句柄
    CONST SMALL_RECT* lpScrollRectangle, // 要滚动或移动的区域
    CONST SMALL_RECT* lpClipRectangle, // 裁剪区域
    COORD dwDestinationOrigin, // 新的位置
    CONST CHAR_INFO* lpFill // 填充字符
);
```

利用这个 API 函数还可以实现删除指定行的操作。下面来举一个例子，程序如下：

```
#include
#include
#include
HANDLE hOut;
void DeleteLine(int row); // 删除一行
void MoveText(int x, int y, SMALL_RECT rc); // 移动文本块区域
void ClearScreen(void); // 清屏
void main()
{
  hOut = GetStdHandle(STD_OUTPUT_HANDLE); // 获取标准输出设备句
柄
  WORD att = FOREGROUND_RED | FOREGROUND_GREEN |
FOREGROUND_INTENSITY |
             BACKGROUND_BLUE ;
  // 背景是蓝色，文本颜色是黄色
  SetConsoleTextAttribute(hOut, att);
  ClearScreen();
  printf("\n\nThe soul selects her own society,\n");
  printf("Then shuts the door;\n");
  printf("On her devine majority;\n");
  printf("Obtrude no more.\n\n");
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  COORD endPos = {0, bInfo.dwSize.Y - 1};
  SetConsoleCursorPosition(hOut, endPos); // 设置光标位置
  SMALL_RECT rc = {0, 2, 40, 5};
  _getch();
  MoveText(10, 5, rc);
  _getch();
  DeleteLine(5);
  CloseHandle(hOut); // 关闭标准输出设备句柄
}

void DeleteLine(int row)
{
  SMALL_RECT rcScroll, rcClip;
  COORD crDest = {0, row - 1};
  CHAR_INFO chFill;
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  rcScroll.Left = 0;
```

```
  rcScroll.Top = row;
  rcScroll.Right = bInfo.dwSize.X - 1;
  rcScroll.Bottom = bInfo.dwSize.Y - 1;
  rcClip = rcScroll;
  chFill.Attributes = bInfo.wAttributes;
  chFill.Char.AsciiChar = ' ';
  ScrollConsoleScreenBuffer(hOut, &rcScroll, &rcClip, crDest,
&chFill);
}

void MoveText(int x, int y, SMALL_RECT rc)
{
  COORD crDest = {x, y};
  CHAR_INFO chFill;
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  chFill.Attributes = bInfo.wAttributes;
  chFill.Char.AsciiChar = ' ';
  ScrollConsoleScreenBuffer(hOut, &rc, NULL, crDest, &chFill);
}

void ClearScreen(void)
{
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  COORD home = {0, 0};
  WORD att = bInfo.wAttributes;
  unsigned long size = bInfo.dwSize.X * bInfo.dwSize.Y;
  FillConsoleOutputAttribute(hOut, att, size, home, NULL);
  FillConsoleOutputCharacter(hOut, ' ', size, home, NULL);
}
```

程序中，实现删除行的操作 DeleteLine 的基本原理是：首先将裁剪区域和移动区域都设置成指定行 row(包括该行)以下的控制台窗口区域，然后将移动的位置指定为(0, row-1)。这样，超出裁剪区域的内容被裁剪掉，从而达到删除行的目的。

需要说明的是，若裁剪区域参数为 NULL，则裁剪区域为整个控制台窗口。

八、光标操作

控制台窗口中的光标反映了文本插入的当前位置，通过 SetConsoleCursorPosition 函数可以改变这个“当前”位置，这样就能控制字符(串)输出。事实上，光标本身的大小和显示或隐藏也可以通过相应的 API 函数进行设定。例如：

```
BOOL SetConsoleCursorInfo( // 设置光标信息
  HANDLE hConsoleOutput, // 句柄
  CONST CONSOLE_CURSOR_INFO *lpConsoleCursorInfo // 光标信息
);

BOOL GetConsoleCursorInfo( // 获取光标信息
  HANDLE hConsoleOutput, // 句柄
  PCONSOLE_CURSOR_INFO lpConsoleCursorInfo // 返回光标信息
);
```

这两个函数都与 CONSOLE_CURSOR_INFO 结构体类型有关，其定义如下：

```
typedef struct _CONSOLE_CURSOR_INFO {
  DWORD dwSize; // 光标百分比大小
  BOOL bVisible; // 是否可见
} CONSOLE_CURSOR_INFO, *PCONSOLE_CURSOR_INFO;
```

需要说明的是，dwSize 值反映了光标的大小，它的值范围为 1-100；当为 1 时，光标最小，仅是一条最靠下的水平细线，当为 100，光标最大，为一个字符大小的方块。

九、读取键盘信息

键盘事件通常有字符事件和按键事件，这些事件所附带的信息构成了键盘信息。它是通过 API 函数 ReadConsoleInput 来获取的，其原型如下：

```
BOOL ReadConsoleInput(
  HANDLE hConsoleInput, // 输入设备句柄
  PINPUT_RECORD lpBuffer, // 返回数据记录
  DWORD nLength, // 要读取的记录数
  LPDWORD lpNumberOfEventsRead // 返回已读取的记录数
);
```

其中，INPUT_RECORD 定义如下：

```
typedef struct _INPUT_RECORD {
  WORD EventType; // 事件类型
  union {
    KEY_EVENT_RECORD KeyEvent;
    MOUSE_EVENT_RECORD MouseEvent;
    WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
    MENU_EVENT_RECORD MenuEvent;
    FOCUS_EVENT_RECORD FocusEvent;
  } Event;
} INPUT_RECORD;
```

与键盘事件相关的记录结构 KEY_EVENT_RECORD 定义如下：

```
typedef struct _KEY_EVENT_RECORD {
  BOOL bKeyDown; // TRUE 表示键按下，FALSE 表示键释放
  WORD wRepeatCount; // 按键次数
  WORD wVirtualKeyCode; // 虚拟键代码
  WORD wVirtualScanCode; // 虚拟键扫描码
  union {
    WCHAR UnicodeChar; // 宽字符
    CHAR AsciiChar; // ASCII 字符
  } uChar; // 字符
  DWORD dwControlKeyState; // 控制键状态
} KEY_EVENT_RECORD;
```

我们知道，键盘上每一个有意义的键都对应着一个唯一的扫描码，虽然扫描码可以作为键的标识，但它依赖于具体设备的。因此，在应用程序中，使用的往往是与具体设备无关的虚拟键代码。这种虚拟键代码是与设备无关的键盘编码。在 Visual C++中，最常用的虚拟键代码已被定义在 Winuser.h 中，例如：VK_SHIFT 表示 SHIFT 键，VK_F1 表示功能键 F1 等。

上述结构定义中，dwControlKeyState 用来表示控制键状态，它可以是 CAPSLOCK_ON(CAPS LOCK 灯亮)、ENHANCED_KEY(按下扩展键)、LEFT_ALT_PRESSED(按下左 ALT 键)、LEFT_CTRL_PRESSED(按下左 CTRL 键)、NUMLOCK_ON (NUM LOCK 灯亮)、RIGHT_ALT_PRESSED(按下右 ALT 键)、RIGHT_CTRL_PRESSED(按下右 CTRL 键)、SCROLLLOCK_ON(SCROLL LOCK 灯亮)和

SHIFT_PRESSED(按下 SHIFT 键)中的一个或多个值的组合。

下面的程序是将用户按键的字符输入到一个控制台窗口的某个区域中，并当按下 NUM LOCK、CAPS LOCK 和 SCROLL LOCK 键时，在控制台窗口的最后一行显示这些键的状态。

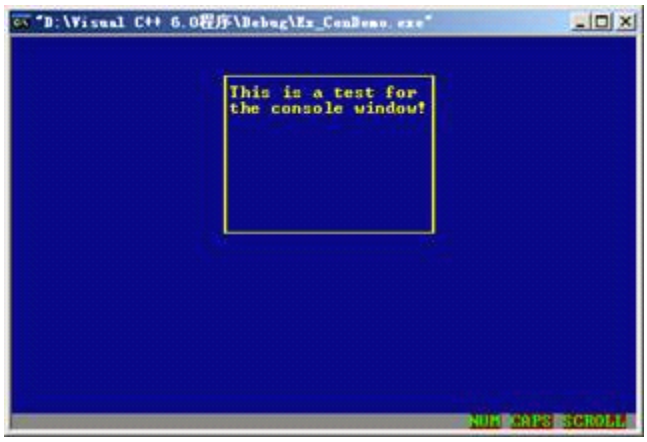
```
#include
HANDLE hOut;
HANDLE hIn;
void DrawBox(bool bSingle, SMALL_RECT rc);
void ClearScreen(void);
void CharWindow(char ch, SMALL_RECT rc); // 将 ch 输入到指定的窗口中
void ControlStatus(DWORD state); // 在最后一行显示控制键的状态
void DeleteTopLine(SMALL_RECT rc); // 删除指定窗口中最上面的行并滚动
void main()
{
  hOut = GetStdHandle(STD_OUTPUT_HANDLE); // 获取标准输出设备句柄
  hIn = GetStdHandle(STD_INPUT_HANDLE); // 获取标准输入设备句柄
  WORD att = FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_INTENSITY |
             BACKGROUND_BLUE ;
  // 背景是蓝色，文本颜色是黄色
  SetConsoleTextAttribute(hOut, att);
  ClearScreen(); // 清屏
  INPUT_RECORD keyRec;
  DWORD state = 0, res;
  char ch;
  SMALL_RECT rc = {20, 2, 40, 12};
  DrawBox(true, rc);
  COORD pos = {rc.Left+1, rc.Top+1};
  SetConsoleCursorPosition(hOut, pos); // 设置光标位置
  for(;;) // 循环
  {
    ReadConsoleInput(hIn, &keyRec, 1, &res);
    if (state != keyRec.Event.KeyEvent.dwControlKeyState) {
      state = keyRec.Event.KeyEvent.dwControlKeyState;
      ControlStatus(state);
    }
```

```
    if (keyRec.EventType == KEY_EVENT){
      if (keyRec.Event.KeyEvent.wVirtualKeyCode == VK_ESCAPE)
break;
      // 按 ESC 键退出循环
      if (keyRec.Event.KeyEvent.bKeyDown) {
        ch = keyRec.Event.KeyEvent.uChar.AsciiChar;
        CharWindow(ch, rc);
      }
    }
  }
  pos.X = 0; pos.Y = 0;
  SetConsoleCursorPosition(hOut, pos); // 设置光标位置
  CloseHandle(hOut); // 关闭标准输出设备句柄
  CloseHandle(hIn); // 关闭标准输入设备句柄
}

void CharWindow(char ch, SMALL_RECT rc) // 将 ch 输入到指定的窗口
中
{
  static COORD chPos = {rc.Left+1, rc.Top+1};
  SetConsoleCursorPosition(hOut, chPos); // 设置光标位置
  if ((ch<0x20)||(ch>0x7e)) return;
  WriteConsoleOutputCharacter(hOut, &ch, 1, chPos, NULL);
  if (chPos.X>=(rc.Right-1))
  {
    chPos.X = rc.Left;
    chPos.Y++;
  }
  if (chPos.Y>(rc.Bottom-1))
  {
    DeleteTopLine(rc);
    chPos.Y = rc.Bottom-1;
  }
  chPos.X++;
  SetConsoleCursorPosition(hOut, chPos); // 设置光标位置
}

void ControlStatus(DWORD state) // 在最后一行显示控制键的状态
{
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
```

```
  COORD home = {0, bInfo.dwSize.Y-1};
  WORD att0 = BACKGROUND_INTENSITY ;
  WORD att1 = FOREGROUND_GREEN | FOREGROUND_INTENSITY |
BACKGROUND_RED;
  FillConsoleOutputAttribute(hOut, att0, bInfo.dwSize.X, home,
NULL);
  FillConsoleOutputCharacter(hOut, ' ', bInfo.dwSize.X, home,
NULL);
  SetConsoleTextAttribute(hOut, att1);
  COORD staPos = {bInfo.dwSize.X-16,bInfo.dwSize.Y-1};
  SetConsoleCursorPosition(hOut, staPos);
  if (state & NUMLOCK_ON)
  WriteConsole(hOut, "NUM", 3, NULL, NULL);
  staPos.X += 4;
  SetConsoleCursorPosition(hOut, staPos);
  if (state & CAPSLOCK_ON)
  WriteConsole(hOut, "CAPS", 4, NULL, NULL);
  staPos.X += 5;
  SetConsoleCursorPosition(hOut, staPos);
  if (state & SCROLLLOCK_ON)
  WriteConsole(hOut, "SCROLL", 6, NULL, NULL);
  SetConsoleTextAttribute(hOut, bInfo.wAttributes); // 恢复原来
的属性
  SetConsoleCursorPosition(hOut, bInfo.dwCursorPosition); // 恢
复原来的光标位置
}

void DeleteTopLine(SMALL_RECT rc)
{
  COORD crDest;
  CHAR_INFO chFill;
  SMALL_RECT rcClip = rc;
  rcClip.Left++; rcClip.Right--;
  rcClip.Top++; rcClip.Bottom--;
  crDest.X = rcClip.Left;
  crDest.Y = rcClip.Top - 1;
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  chFill.Attributes = bInfo.wAttributes;
  chFill.Char.AsciiChar = ' ';
  ScrollConsoleScreenBuffer(hOut, &rcClip, &rcClip, crDest, &chF
```

```
ill);
}
```

程序运行结果如下图所示：



十、读取鼠标信息

与读取键盘信息方法相似，鼠标信息也是通过 ReadConsoleInput 来获取的，其 MOUSE_EVENT_RECORD 具有下列定义：

```
typedef struct _MOUSE_EVENT_RECORD {
  COORD dwMousePosition; // 当前鼠标位置
  DWORD dwButtonState; // 鼠标按钮状态
  DWORD dwControlKeyState; // 键盘控制键状态
  DWORD dwEventFlags; // 事件状态
} MOUSE_EVENT_RECORD;
```

其中，dwButtonState 反映了用户按下鼠标按钮的情况，它可以是：FROM_LEFT_1ST_BUTTON_PRESSED(最左边按钮)、RIGHTMOST_BUTTON_PRESSED(最右边按钮)、FROM_LEFT_2ND_BUTTON_PRESSED(左起第二个按钮)、FROM_LEFT_3RD_BUTTON_PRESSED(左起第三个按钮)和 FROM_LEFT_4TH_BUTTON_PRESSED (左起第四个按钮)。而 dwEventFlags 表示鼠标的事件，如 DOUBLE_CLICK(双击)、MOUSE_MOVED(移动)和 MOUSE_WHEELED(滚轮滚动，只适用于 Windows 2000/XP)。dwControlKeyState 的含义同前。

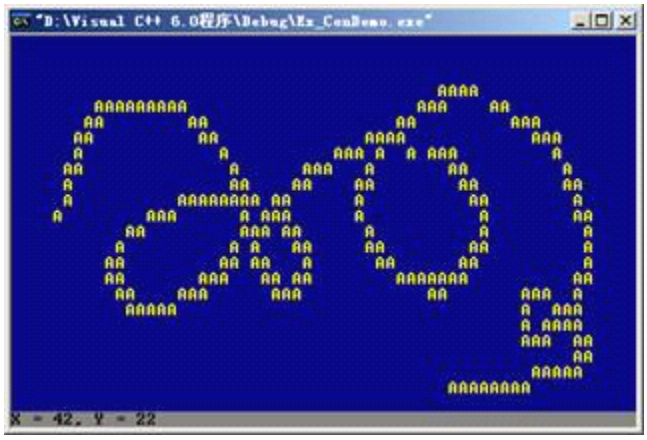
下面举一个例子。这个例子能把鼠标的当前位置显示在控制台窗口的最后一行

上，若单击鼠标左键，则在当前位置处写一个字符‘A’，若双击鼠标任一按钮，则程序终止。具体代码如下：

```
#include <WINDOWS.H>
#include <STDIO.H>
#include <STRING.H>
HANDLE hOut;
HANDLE hIn;
void ClearScreen(void);
void DispMousePos(COORD pos); // 在最后一行显示鼠标位置
void main()
{
  hOut = GetStdHandle(STD_OUTPUT_HANDLE); // 获取标准输出设备句
柄
  hIn = GetStdHandle(STD_INPUT_HANDLE); // 获取标准输入设备句柄
  WORD att = FOREGROUND_RED | FOREGROUND_GREEN |
FOREGROUND_INTENSITY |
             BACKGROUND_BLUE ;
  // 背景是蓝色，文本颜色是黄色
  SetConsoleTextAttribute(hOut, att);
  ClearScreen(); // 清屏
  INPUT_RECORD mouseRec;
  DWORD state = 0, res;
  COORD pos = {0, 0};
  for(;;) // 循环
  {
    ReadConsoleInput(hIn, &mouseRec, 1, &res);
    if (mouseRec.EventType == MOUSE_EVENT){
      if (mouseRec.Event.MouseEvent.dwEventFlags == DOUBLE_CLICK)
 break;
        // 双击鼠标退出循环
        pos = mouseRec.Event.MouseEvent.dwMousePosition;
        DispMousePos(pos);
        if (mouseRec.Event.MouseEvent.dwButtonState ==
FROM_LEFT_1ST_BUTTON_PRESSED)
                    FillConsoleOutputCharacter(hOut, 'A', 1, pos,
 NULL);
     }
    }
    pos.X = 0; pos.Y = 0;
    SetConsoleCursorPosition(hOut, pos); // 设置光标位置
```

```
    CloseHandle(hOut); // 关闭标准输出设备句柄
    CloseHandle(hIn); // 关闭标准输入设备句柄
  }

void DispMousePos(COORD pos) // 在最后一行显示鼠标位置
{
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  COORD home = {0, bInfo.dwSize.Y-1};
  WORD att0 = BACKGROUND_INTENSITY ;
  FillConsoleOutputAttribute(hOut, att0, bInfo.dwSize.X, home,
NULL);
  FillConsoleOutputCharacter(hOut, ' ', bInfo.dwSize.X, home,
NULL);
  char s[20];
  sprintf(s,"X = %2lu, Y = %2lu",pos.X, pos.Y);
  SetConsoleTextAttribute(hOut, att0);
  SetConsoleCursorPosition(hOut, home);
  WriteConsole(hOut, s, strlen(s), NULL, NULL);
  SetConsoleTextAttribute(hOut, bInfo.wAttributes); // 恢复原来
的属性
  SetConsoleCursorPosition(hOut, bInfo.dwCursorPosition); // 恢
复原来的光标位置
}

void ClearScreen(void)
{
  CONSOLE_SCREEN_BUFFER_INFO bInfo;
  GetConsoleScreenBufferInfo( hOut, &bInfo );
  COORD home = {0, 0};
  unsigned long size = bInfo.dwSize.X * bInfo.dwSize.Y;
  FillConsoleOutputAttribute(hOut, bInfo.wAttributes, size, home,
 NULL);
  FillConsoleOutputCharacter(hOut, ' ', size, home, NULL);
}
```

程序运行结果如下：



十一、结语

综上所述，利用控制台窗口的 Widows API 函数可以设计简洁美观的文本界面，使得用 Visual C++ 6.0 开发环境深入学习 C++以及文本界面设计成为一件比较容易的事件。当然文本界面的设计还需要一定的方法和技巧，限于篇幅，这里不再阐述。