# 这一节我们来学习文本编程

## 有关字体的信息图解

|  |
| --- |
|  |

## 1，新建一个MFC工程，取名：text\_appl，用来学习文本编辑器程序的编程

|  |
| --- |
|  |

## 2.我们编译运行一下这个程序，效果如下

|  |
| --- |
|  |

### 我们发现，它并没有编辑框功能，那么我们应该怎么办呢？我们需要创建一个插入符，使用的函数是CWnd::CreateSolidCaret

## 3.给view类添加一个WM\_CREATE消息的响应函数，在类上面点击右键-》类向导-》消息，类选择view类，消息选择WM\_CREAT，点击右边的添加处理程序按钮，点击应用，如图

|  |
| --- |
|  |

### 点击确定

|  |
| --- |
|  |

## 4.我们在这个函数里面添加创建插入符的代码，注意创建后还需要让他显示处理使用ShowCaret函数

|  |
| --- |
|  |

### 运行程序，发现插入符出来了，

|  |
| --- |
|  |

### 但是你会发现你，此时你输入文本是没有效果的。那是为什么？

## 5.其实插入符的大小是根据当前字体的大小来确定的，字体越大。插入符的大小就越大。可是我们如何获取当前字体的大小？可以使用CDC::GetTextMetrics函数，我们修改一下代码，

|  |
| --- |
|  |

## 6.其实我们还可以创建图形插入符，使用CWnd::CreateCaret(CBitmap\* *pBitmap*),他需要一个位图，可以点击项目-添加资源来插入一张bmp格式的图片

|  |
| --- |
|  |
|  |
|  |

### 然后位图就导进来了

|  |
| --- |
|  |

## 7.创建一个位图对象，然后利用它来加载位图资源

|  |
| --- |
| int CtextapplView::OnCreate(LPCREATESTRUCT lpCreateStruct)  {  if (CView::OnCreate(lpCreateStruct) == -1)  return -1;  // TODO: 在此添加您专用的创建代码  // 创建客户DC  CClientDC dc(this);  //定义一个文本度量结构体的对象  TEXTMETRIC tm;  //用dc对象来获取文本度量信息填充文本度量结构体对象  dc.GetTextMetrics(&tm);  //创建普通插入符  //CreateSolidCaret(tm.tmAveCharWidth/8, tm.tmHeight);  //创建位图插入符  CBitmap bitmap;  bitmap.LoadBitmapW(MAKEINTRESOURCE(IDB\_BITMAP1));  CreateCaret(&bitmap);//这个是CWnd类的函数不是CDC的函数  //显示插入符  ShowCaret();  return 0;  } |

### 运行程序，发现根本没有插入符，这是为什么？

|  |
| --- |
|  |

## 8.因为这里的Bitmap对象是一个局部变量，当程序运行到大括号外面，它的生命周期结束，被销毁了。正确的做法是把这个bitmap作为view类的成员变量

|  |
| --- |
|  |

### 再次编译运行程序，位图插入符出现了

|  |
| --- |
|  |

## 9.下面我们来输出一串文字，需要在view类的OnDraw函数里面添加代码，这里我们使用到一个MFC的字符串类CString，它的功能很强大，它重载了=，+，+=操作符，还封装了win32 平台sdk的LoadString函数，可以直接从资源文件里面加载字符串

|  |
| --- |
|  |
|  |
|  |
| // example for CString::CString  CString s1;                    // Empty string  CString s2( "cat" );           // From a C string literal  CString s3 = s2;               // Copy constructor  CString s4( s2 + " " + s3 );   // From a string expression  CString s5( 'x' );             // s5 = "x"  CString s6( 'x', 6 );          // s6 = "xxxxxx"  CString s7((LPCSTR)ID\_FILE\_NEW); // s7 = "Create a new document"  CString city = "Philadelphia"; // NOT the assignment operator |

### 我们在资源文件的string table里面新建一个字符串资源，

|  |
| --- |
|  |

### 我们使用CString的LoadString方法来加载到窗口

|  |
| --- |
|  |

### 编译运行程序，效果如下：

|  |
| --- |
|  |

## 10.下面我们来学习一下一个叫做路径层的概念，这里涉及到两个API，BeginPath和EndPath，我们可以用路径层把我们输出的文字圈起来。需要包裹文字的话，我们需要知道文字的左上角和右下角的坐标，左上角我们知道，右下角怎么获取？需要使用一个API函数叫做CDC::GetTextExtent，他有2个重载函数

|  |
| --- |
|  |

## 11.为了更好的演示效果，我们先把代码改为下面的样子

|  |
| --- |
|  |

### 此时效果是这样子的

|  |
| --- |
|  |

## 12.下面我们来实现路径层把上面的文本包裹起来，代码如下

|  |
| --- |
|  |

### 但是，当我们编译运行程序，看不到任何效果

|  |
| --- |
|  |

### 那么问题来了，这个路径层究竟有什么用？

## 13.要测试路径层的作用，我们可以绘制一些线条来覆盖第一行文本，看看有什么效果

|  |
| --- |
|  |

### 编译运行程序，我们仍然看不出有什么好处

|  |
| --- |
|  |

## 14，只是使用路径层并没有作用，我们需要配合一个API：

### CDC::SelectClipPath，语法如下：

|  |
| --- |
|  |

## 15.这里我们可以使用RGN\_DIFF模式把路径层排除在外，这样子，上面绘制的线条就不会覆盖路径层

|  |
| --- |
|  |

### 运行程序，发现我们绘制的直线没有绘制到路径层里面的文字上面

|  |
| --- |
|  |

## 16.我们将模式改为RGN\_AND

|  |
| --- |
|  |

### 编译运行程序，效果如下

|  |
| --- |
|  |

### 可见这两种模式的效果是相反的

## 17.我们把模式改为RGN\_OR

|  |
| --- |
|  |

### 没有任何效果，

|  |
| --- |
|  |

## 18.模式改为RGN\_XOR

|  |
| --- |
|  |

### 也是有效果的

|  |
| --- |
|  |

### 可见，我们在作图的时候可以使用路径层来实现某一部分和其他部分效果不一样的功能

## 19.我们修改一下代码想给下面的代码也做一个路径层，也就是想要2个路径层，

|  |
| --- |
|  |

### 效果如下。

|  |
| --- |
|  |

### 可见是可以使用多个路径层的，但是要代码书写注意顺序，写错了就没有这个效果

# 下面我们来学习类似于编辑框的文本输出功能

## 实现思想：原则上是当我们输入的时候我们需要响应WM\_CHAR消息，用TextOut函数输出我们按下的每一个字符，但是，这个函数每一次输出都需要我们获取字母的坐标，由于每一个字母的宽度其实都是不一样的，这样子很难获取，所以我们使用一个累加的方法，我们先定义一个空字符串，在每一次的WM\_CHAR消息响应中。我们把按下的字母累加到这个字符串中然后输出整个字符串，这样子虽然效率比较低，但是它可以解决我们的问题。

## 20.给view类添加一个WM\_CHAR消息响应函数

|  |
| --- |
|  |
|  |

## 21.需要给view类增加一个CString类型的成员变量，用来保存我们按下的字符。

|  |
| --- |
|  |
|  |
|  |

## 22.先在view类的构造函数中把这个CString成员变量初始化为空，

|  |
| --- |
|  |

## 23.我们来实现文本输入功能，但是在这之前，这里需要注意一个问题，当我们在窗口客户区点击鼠标左键，插入符应该出现在鼠标点击的地方。所以我们需要给view类添加一个鼠标左键按下的消息响应函数。

|  |
| --- |
|  |
|  |

## 24.那么我们如何来移动插入符？这里有一个CWnd::SetCaretPos(POINT *point*)函数，我们可以使用它来实现移动插入符，注意，这是一个静态函数。

|  |
| --- |
|  |

## 25，我们只需要在鼠标左键按下消息处理函数里面调用这个函数即可

|  |
| --- |
|  |

### 运行程序效果如下

|  |
| --- |
|  |
|  |
|  |

## 26.其实还没完，当我们输入完一行字符，我们鼠标左键按下的时候，我们的保存按下字符的成员变量需要把先前保存的所有字符清空

|  |
| --- |
|  |

## 27而且因为TextOut函数需要字符串的坐标，所以当鼠标左键按下的时候，我们需要用一个成员变量把鼠标点击位置的当前坐标保存起来。我们给view类添加一个CPoint类型的需要变量，并且在view类构造函数里面把它初始化为0。

|  |
| --- |
|  |
|  |
|  |

## 28.在鼠标左键按下的处理函数中把当前位置的点赋值给m\_prOrg成员变量

|  |
| --- |
|  |

## 29.在OnChar函数中处理文本输出，注意：当我们按下的是回车键，我们的m\_strLine需要清空里面保存的所有字符。所以首先我们需要判断按下的是否是回车，回车的ascII码是13，需要清空，如何需要另起一行，横坐标是没有变化的纵坐标需要在原来的基础上加上一个当前字体的高度，可以使用CDC::GetTextMetric获得，所以我们需要创建一个DC对象。按下回车键的处理代码如下

|  |
| --- |
|  |

## 30，接着就是处理退格键的按下，当我们按下退格键，字符串需要把最后面的字符删除，然后在原来的起点先把原来的文本改为和背景颜色一样，然后重新输出新字符串，那么如何获取窗口的背景颜色？需要一个API，叫做CDC::GetBkColor ，具体用法如下

|  |
| --- |
|  |

### 设置文本颜色的API是CDC::SetTextColor，语法如下

|  |
| --- |
|  |

### 注意，这个API会返回设置文本颜色之前的文本颜色，也就是它原来的颜色，好处是方便我们把原来的颜色设置回来。

## 31.我们下面来完成这个功能，先把原来的文本用背景色输出，也就是覆盖掉，然后把字符串的最后一个字符删除，需要使用CString类的Left方法取最后一个字符左边的所有字符，然后需要把文本原来的颜色设置回来

|  |
| --- |
|  |

## 32.处理了这两种情况后，就是普通字符的处理，很简单，就是把它添加到字符串变量中保存起来,然后就把文本输出，我们的代码如下

|  |
| --- |
| **void CtextapplView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)**  **{**  **// TODO: 在此添加消息处理程序代码和/或调用默认值**  **CClientDC dc(this);**  **TEXTMETRIC tm;**  **dc.GetTextMetrics(&tm);**  **if (0x0d == nChar)**  **{ //按了回车键，m\_strLine需要清空内容**  **m\_strLine.Empty();**  **m\_ptOrg.y += tm.tmHeight;//按下回车后，下一行的纵坐标需要在原来的继承上加上当前字体的高度tm.tmHeight**  **}//下面需要处理退格键按下**  **else if (0x08 == nChar)**  **{**  **COLORREF clr = dc.SetTextColor(dc.GetBkColor());//把文本颜色设置为背景颜色，并且保存先前的颜色**  **dc.TextOutW(m\_ptOrg.x, m\_ptOrg.y, m\_strLine);//用背景颜色输出文本就相当于擦除文本**  **//把字符串成员最后一个字符删除,这里使用一个技巧：用CString的Left方法来获取最后一个字符前面的使用字符**  **m\_strLine = m\_strLine.Left(m\_strLine.GetLength() - 1);**  **//把文本原来的颜色设置回来**  **dc.SetTextColor(clr);**  **}**  **else**  **{**  **m\_strLine += (WCHAR)nChar;**  **}**  **//输出文本**  **dc.TextOutW(m\_ptOrg.x, m\_ptOrg.y, m\_strLine);**  **CView::OnChar(nChar, nRepCnt, nFlags);**  **}** |

### 编译运行程序，发现的确可以输入和使用退格，回车键的功能了

|  |
| --- |
|  |

### 而且插入符会移动到左键点击的位置

|  |
| --- |
|  |

### 但是问题还是很明显的，就是当文本输入后插入符没有往后移动

## 33.我们可以先获取我们的字符串的显示宽度，使用CDC::GetTextExtent函数，然后计算插入符的位置，然后使用SetCaretPos函数设置插入符的位置，注意需要在输出文本之前设置好。代码如下

|  |
| --- |
| void CtextapplView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)  {  // TODO: 在此添加消息处理程序代码和/或调用默认值  CClientDC dc(this);  TEXTMETRIC tm;  dc.GetTextMetrics(&tm);  if (0x0d == nChar)  { //按了回车键，m\_strLine需要清空内容  m\_strLine.Empty();  m\_ptOrg.y += tm.tmHeight;//按下回车后，下一行的纵坐标需要在原来的继承上加上当前字体的高度tm.tmHeight  }//下面需要处理退格键按下  else if (0x08 == nChar)  {  COLORREF clr = dc.SetTextColor(dc.GetBkColor());//把文本颜色设置为背景颜色，并且保存先前的颜色  dc.TextOutW(m\_ptOrg.x, m\_ptOrg.y, m\_strLine);//用背景颜色输出文本就相当于擦除文本  //把字符串成员最后一个字符删除,这里使用一个技巧：用CString的Left方法来获取最后一个字符前面的使用字符  m\_strLine = m\_strLine.Left(m\_strLine.GetLength() - 1);  //把文本原来的颜色设置回来  dc.SetTextColor(clr);  }  else  {  m\_strLine += (WCHAR)nChar;  }  //输出文本  //1.获取屏幕文本的显示宽度  CSize sz = dc.GetTextExtent(m\_strLine);//获取屏幕文本显示的尺寸包括宽度和高度  CPoint pt;  pt.x = m\_ptOrg.x + sz.cx;  pt.y = m\_ptOrg.y;  //先设置插入符的位置，需要在输出文本之前设置  SetCaretPos(pt);  dc.TextOutW(m\_ptOrg.x, m\_ptOrg.y, m\_strLine);      CView::OnChar(nChar, nRepCnt, nFlags);  } |

### 这样子的效果就对了，当然其实我们可以把插入符改为普通的插入符，这里先不改

|  |
| --- |
|  |

# 下面我们来学习修改字体

## 我们的程序字体不太好看，我们想修改一下字体，MFC中提供了一个CFont类，封装了字体相关的方法。CFont是CGdiObject的子类。CFont对象构造后，不能直接使用，必须调用他的一下成员函数如：CreateFont，CreateFontIndirect或者CreatePointFont函数来进行初始化

|  |
| --- |
|  |

## 我们这里学习CreatePointFont函数

|  |
| --- |
|  |

## 字体可以在vs里面查看，点击工具-》选项-》字体和颜色，效果如下

|  |
| --- |
|  |

## 我们也可以在c盘里面：C:\Windows\Fonts查看，不过我们的程序并不能够支持所有的字体，具体支持那些字体可以通过试验来确定。

## 34.我们在OnChar函数的开头部分创建字体。注意创建完字体后，需要把它选进设备描述表同时需要保存原来的字体，使用完毕后，我们需要把原来的字体选入设备描述表方便其他程序使用

## 其实，我们没有不要自己从零开始写字处理软件，MFC通过了CEdit类他默认就有编辑功能，我们主要是想学习一下相关API。还有CEditView和CRichEditView类。

# 下面我们来学习一个有趣的功能：就是KTV字幕变色功能

## 要实现这个功能，用上面的TextOut函数效果不太好，我们可以使用另外一个API：CDC::DrawText ，他的用法如下

|  |
| --- |
|  |
| *nFormat*  Specifies the method of formatting the text. It can be any combination of the following values (combine using the bitwise OR operator):   * **DT\_BOTTOM**   Specifies bottom-justified text. This value must be combined with **DT\_SINGLELINE**. * **DT\_CALCRECT**   Determines the width and height of the rectangle. If there are multiple lines of text, **DrawText** will use the width of the rectangle pointed to by *lpRect* and extend the base of the rectangle to bound the last line of text. If there is only one line of text, **DrawText** will modify the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawText** returns the height of the formatted text, but does not draw the text. * **DT\_CENTER**   Centers text horizontally. * **DT\_END\_ELLIPSIS** or **DT\_PATH\_ELLIPSIS**Replaces part of the given string with ellipses, if necessary, so that the result fits in the specified rectangle. The given string is not modified unless the **DT\_MODIFYSTRING** flag is specified.   You can specify **DT\_END\_ELLIPSIS** to replace characters at the end of the string, or **DT\_PATH\_ELLIPSIS** to replace characters in the middle of the string. If the string contains backslash (\) characters, **DT\_PATH\_ELLIPSIS** preserves as much as possible of the text after the last backslash.   * **DT\_EXPANDTABS**   Expands tab characters. The default number of characters per tab is eight. * **DT\_EXTERNALLEADING**   Includes the font’s external leading in the line height. Normally, external leading is not included in the height of a line of text. * **DT\_LEFT**   Aligns text flush-left. * **DT\_MODIFYSTRING**Modifies the given string to match the displayed text. This flag has no effect unless the **DT\_END\_ELLIPSIS** or **DT\_PATH\_ELLIPSIS** flag is specified.   **Note** Some *uFormat* flag combinations can cause the passed string to be modified. Using **DT\_MODIFYSTRING** with either **DT\_END\_ELLIPSIS** or **DT\_PATH\_ELLIPSIS** may cause the string to be modified, causing an assertion in the **CString** override.   * **DT\_NOCLIP**   Draws without clipping. **DrawText** is somewhat faster when **DT\_NOCLIP** is used. * **DT\_NOPREFIX**   Turns off processing of prefix characters. Normally, **DrawText** interprets the ampersand (**&**) mnemonic-prefix character as a directive to underscore the character that follows, and the two-ampersand (**&&**) mnemonic-prefix characters as a directive to print a single ampersand. By specifying **DT\_NOPREFIX**, this processing is turned off. * **DT\_PATH\_ELLIPSIS** * **DT\_RIGHT**   Aligns text flush-right. * **DT\_SINGLELINE**   Specifies single line only. Carriage returns and linefeeds do not break the line. * **DT\_TABSTOP**   Sets tab stops. The high-order byte of *nFormat* is the number of characters for each tab. The default number of characters per tab is eight. * **DT\_TOP**   Specifies top-justified text (single line only). * **DT\_VCENTER**   Specifies vertically centered text (single line only). * **DT\_WORDBREAK**   Specifies word-breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by *lpRect*. A carriage return–linefeed sequence will also break the line.   **Note** The values **DT\_CALCRECT**, **DT\_EXTERNALLEADING**, **DT\_INTERNAL**, **DT\_NOCLIP**, and **DT\_NOPREFIX** cannot be used with the **DT\_TABSTOP** value.  **Remarks**  Call this member function to format text in the given rectangle. It formats text by expanding tabs into appropriate spaces, aligning text to the left, right, or center of the given rectangle, and breaking text into lines that fit within the given rectangle. The type of formatting is specified by *nFormat*.  This member function uses the device context’s selected font, text color, and background color to draw the text. Unless the **DT\_NOCLIP** format is used, **DrawText** clips the text so that the text does not appear outside the given rectangle. All formatting is assumed to have multiple lines unless the **DT\_SINGLELINE** format is given.  If the selected font is too large for the specified rectangle, the **DrawText** member function does not attempt to substitute a smaller font.  If the **DT\_CALCRECT** flag is specified, the rectangle specified by *lpRect* will be updated to reflect the width and height needed to draw the text.  If the **TA\_UPDATECP** text-alignment flag has been set (see [CDC::SetTextAlign](mk:@MSITStore:C:\Program%20Files%20(x86)\Microsoft%20Visual%20Studio\MSDN98\98VS\2052\vcmfc.chm::/html/_mfc_cdc.3a3a.settextalign.htm)), **DrawText** will display text starting at the current position, rather than at the left of the given rectangle. **DrawText** will not wrap text when the **TA\_UPDATECP** flag has been set (that is, the **DT\_WORDBREAK** flag will have no effect). |

### 这个API会把文字的输出局限在一个矩形之内，进入文本比矩形的宽度还长，超出的部分被截断，此外我们还需要一个定时器，设置他在一个时间间隔里面改变矩形的宽度。这里我们需要用到一个API叫做：CWnd::SetTimer，它比较有意思，第一个参数，我们可以给他一个整数作为标识，第二个参数是时间间隔，第三个参数是回调函数，也可以传递NULL，每一次出发定时器，如果有回调函数，系统就会调用你的回调函数，如果是NULL，系统就会发送一个WM\_TIMER消息到消息队列中，我们可以添加对这个消息的处理函数来处理这个消息。在实际开发中，经常传递NULL给第三个参数。

|  |
| --- |
|  |

## 35.我们在view类的OnCreate函数里面调用SetTimer方法

|  |
| --- |
|  |

## 36.我们给view类添加一个WM\_TIMER的消息处理函数

|  |
| --- |
|  |
|  |

### 注意到这个函数有一个nIDEvent这个就是系统传递过来的定时器ID，因为可能有多个定时器，我们只处理我们感兴趣的定时器在这里是id为1的定时器这个是我们设置的定时器。

## 37.我们需要在这个处理函数里面对我们感兴趣的定时器事件进行响应，不过我们的程序只设置了一个定时器，我们就没有必要再做判断。我们需要一个成员变量来改变DrawText函数里面的进行的宽度，在view类上面点击右键-》添加-》成员变量，然后填写变量消息，设置为私有，点击确定

|  |
| --- |
|  |
|  |

## 38.然后我们需要在view类的构造函数里面初始化这个成员变量

|  |
| --- |
|  |

## 39.我们开始实现WM\_TIMER事件的处理代码，我们想实现的功能是每隔100毫秒显示文本的矩形区域的宽度会增加5给像素，这样子他就会越来越宽，有KTV的效果，我们进入OnTimer函数，添加如下代码

|  |
| --- |
|  |

### 效果，

|  |
| --- |
|  |
|  |

## 40.这个程序有点问题，当我们把输出文本API的对齐方式改为DT\_RIGHT;你会发现所有字符都填满后，它还会继续往右移动，这是不对的，我们需要让他从新开始才最左边输出，我们来解决这个问题，

|  |
| --- |
|  |

### 效果：

|  |
| --- |
|  |

### ok，这就对了。