`CView` Class

Article • 05/16/2022 • 29 minutes to read

Provides the basic functionality for user-defined view classes.

Syntax

```
class AFX_NOVTABLE CView : public CWnd
```

Members

Protected Constructors

Name	Description
CView::CView	Constructs a `CView` object.

Public Methods

Name	Description
CView::DoPreparePrinting	Displays Print dialog box and creates printer device context; call when overriding the `OnPreparePrinting` member function.
CView::GetDocument	Returns the document associated with the view.
CView::IsSelected	Tests whether a document item is selected. Required for OLE support.
CView::OnDragEnter	Called when an item is first dragged into the drag-and-drop region of a view.
CView::OnDragLeave	Called when a dragged item leaves the drag-and-drop region of a view.
CView::OnDragOver	Called when an item is dragged over the drag-and-drop region of a view.

Name	Description
CView::OnDragScroll	Called to determine whether the cursor is dragged into the scroll region of the window.
CView::OnDrop	Called when an item has been dropped into the drag-and-drop region of a view, default handler.
CView::OnDropEx	Called when an item has been dropped into the drag-and-drop region of a view, primary handler.
CView::OnInitialUpdate	Called after a view is first attached to a document.
CView::OnPrepareDC	Called before the `OnDraw` member function is called for screen display or the `OnPrint` member function is called for printing or print preview.
CView::OnScroll	Called when OLE items are dragged beyond the borders of the view.
CView::OnScrollBy	Called when a view containing active in-place OLE items is scrolled.

Protected Methods

Name	Description
CView::OnActivateFrame	Called when the frame window containing the view is activated or deactivated.
CView::OnActivateView	Called when a view is activated.
CView::OnBeginPrinting	Called when a print job begins; override to allocate graphics device interface (GDI) resources.
CView::OnDraw	Called to render an image of the document for screen display, printing, or print preview. Implementation required.
CView::OnEndPrinting	Called when a print job ends; override to deallocate GDI resources.
CView::OnEndPrintPreview	Called when preview mode is exited.
CView::OnPreparePrinting	Called before a document is printed or previewed; override to initialize Print dialog box.
CView::OnPrint	Called to print or preview a page of the document.
CView::OnUpdate	Called to notify a view that its document has been modified.

Remarks

A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen or printer and interprets user input as operations upon the document.

A view is a child of a frame window. More than one view can share a frame window, as in the case of a splitter window. The relationship between a view class, a frame window class, and a document class is established by a `CDocTemplate` object. When the user opens a new window or splits an existing one, the framework constructs a new view and attaches it to the document.

A view can be attached to only one document, but a document can have multiple views attached to it at once — for example, if the document is displayed in a splitter window or in multiple child windows in a multiple document interface (MDI) application. Your application can support different types of views for a given document type; for example, a word-processing program might provide both a complete text view of a document and an outline view that shows only the section headings. These different types of views can be placed in separate frame windows or in separate panes of a single frame window if you use a splitter window.

A view may be responsible for handling several different types of input, such as keyboard input, mouse input or input via drag-and-drop, as well as commands from menus, toolbars, or scroll bars. A view receives commands forwarded by its frame window. If the view does not handle a given command, it forwards the command to its associated document. Like all command targets, a view handles messages via a message map.

The view is responsible for displaying and modifying the document's data but not for storing it. The document provides the view with the necessary details about its data. You can let the view access the document's data members directly, or you can provide member functions in the document class for the view class to call.

When a document's data changes, the view responsible for the changes typically calls the CDocument::UpdateAllViews function for the document, which notifies all the other views by calling the `OnUpdate` member function for each. The default implementation of `OnUpdate` invalidates the view's entire client area. You can override it to invalidate only those regions of the client area that map to the modified portions of the document.

To use `CView`, derive a class from it and implement the `OnDraw` member function to perform screen display. You can also use `OnDraw` to perform printing and print preview. The framework handles the print loop for printing and previewing your document.

A view handles scroll-bar messages with the CWnd::OnHScroll and CWnd::OnVScroll member functions. You can implement scroll-bar message handling in these functions, or you can use the `CView` derived class CScrollView to handle scrolling for you.

Besides `CScrollView`, the Microsoft Foundation Class Library provides nine other classes derived from `CView`:

- CCtrlView, a view that allows usage of document - view architecture with tree, list, and rich edit controls.
- CDaoRecordView, a view that displays database records in dialog-box controls.
- CEditView, a view that provides a simple multiline text editor. You can use a `CEditView` object as a control in a dialog box as well as a view on a document.
- CFormView, a scrollable view that contains dialog-box controls and is based on a dialog template resource.
- CListView, a view that allows usage of document - view architecture with list controls.
- CRecordView, a view that displays database records in dialog-box controls.
- CRichEditView, a view that allows usage of document - view architecture with rich edit controls.
- CScrollView, a view that automatically provides scrolling support.
- CTreeView, a view that allows usage of document - view architecture with tree controls.

The `CView` class also has a derived implementation class named `CPreviewView`, which is used by the framework to perform print previewing. This class provides support for the features unique to the print-preview window, such as a toolbar, single- or double-page preview, and zooming, that is, enlarging the previewed image. You don't need to call or override any of `CPreviewView`'s member functions unless you want to implement your own interface for print preview (for example, if you want to support editing in print preview mode). For more information on using `CView`, see Document/View Architecture and Printing. In addition, see Technical Note 30 for more details on customizing print preview.

Inheritance Hierarchy

CObject

CCmdTarget

CWnd

`CView`

Requirements

Header: `afxwin.h`

`CView::CView`

Constructs a `CView` object.

```
CView();
```

Remarks

The framework calls the constructor when a new frame window is created or a window is split. Override the OnInitialUpdate member function to initialize the view after the document is attached.

`CView::DoPreparePrinting`

Call this function from your override of OnPreparePrinting to invoke the Print dialog box and create a printer device context.

```
BOOL DoPreparePrinting(CPrintInfo* pInfo);
```

Parameters

pInfo
Points to a CPrintInfo structure that describes the current print job.

Return Value

Nonzero if printing or print preview can begin; 0 if the operation has been canceled.

Remarks

This function's behavior depends on whether it is being called for printing or print preview (specified by the `m_bPreview` member of the *pInfo* parameter). If a file is being printed, this function invokes the Print dialog box, using the values in the CPrintInfo structure that *pInfo* points to; after the user has closed the dialog box, the function creates a printer device context based on settings the user specified in the dialog box and returns this device context through the *pInfo* parameter. This device context is used to print the document.

If a file is being previewed, this function creates a printer device context using the current printer settings; this device context is used for simulating the printer during preview.

`CView::GetDocument`

Call this function to get a pointer to the view's document.

```
CDocument* GetDocument() const;
```

Return Value

A pointer to the CDocument object associated with the view. `NULL` if the view is not attached to a document.

Remarks

This allows you to call the document's member functions.

`CView::IsSelected`

Called by the framework to check whether the specified document item is selected.

```
virtual BOOL IsSelected(const CObject* pDocItem) const;
```

Parameters

`pDocItem`
Points to the document item being tested.

Return Value

Nonzero if the specified document item is selected; otherwise 0.

Remarks

The default implementation of this function returns `FALSE`. Override this function if you are implementing selection using CDocItem objects. You must override this function if your view contains OLE items.

`CView::OnActivateFrame`

Called by the framework when the frame window containing the view is activated or deactivated.

```
virtual void OnActivateFrame(
    UINT nState,
    CFrameWnd* pFrameWnd);
```

Parameters

nState
Specifies whether the frame window is being activated or deactivated. It can be one of the following values:

- `WA_INACTIVE` The frame window is being deactivated.
- `WA_ACTIVE` The frame window is being activated through some method other than a mouse click (for example, by use of the keyboard interface to select the window).
- `WA_CLICKACTIVE` The frame window is being activated by a mouse click

pFrameWnd
Pointer to the frame window that is to be activated.

Remarks

Override this member function if you want to perform special processing when the frame window associated with the view is activated or deactivated. For example, CFormView performs this override when it saves and restores the control that has focus.

`CView::OnActivateView`

Called by the framework when a view is activated or deactivated.

```
virtual void OnActivateView(
    BOOL bActivate,
    CView* pActivateView,
    CView* pDeactiveView);
```

Parameters

bActivate

Indicates whether the view is being activated or deactivated.

pActivateView

Points to the view object that is being activated.

pDeactiveView

Points to the view object that is being deactivated.

Remarks

The default implementation of this function sets the focus to the view being activated. Override this function if you want to perform special processing when a view is activated or deactivated. For example, if you want to provide special visual cues that distinguish the active view from the inactive views, you would examine the *bActivate* parameter and update the view's appearance accordingly.

The *pActivateView* and *pDeactiveView* parameters point to the same view if the application's main frame window is activated with no change in the active view — for example, if the focus is being transferred from another application to this one, rather than from one view to another within the application or when switching amongst MDI child windows. This allows a view to re-realize its palette, if needed.

These parameters differ when CFrameWnd::SetActiveView is called with a view that is different from what CFrameWnd::GetActiveView would return. This happens most often with splitter windows.

`CView::OnBeginPrinting`

Called by the framework at the beginning of a print or print preview job, after `OnPreparePrinting` has been called.

```
virtual void OnBeginPrinting(
    CDC* pDC,
    CPrintInfo* pInfo);
```

Parameters

pDC
Points to the printer device context.

pInfo
Points to a CPrintInfo structure that describes the current print job.

Remarks

The default implementation of this function does nothing. Override this function to allocate any GDI resources, such as pens or fonts, needed specifically for printing. Select the GDI objects into the device context from within the OnPrint member function for each page that uses them. If you are using the same view object to perform both screen display and printing, use separate variables for the GDI resources needed for each display; this allows you to update the screen during printing.

You can also use this function to perform initializations that depend on properties of the printer device context. For example, the number of pages needed to print the document may depend on settings that the user specified from the Print dialog box (such as page length). In such a situation, you cannot specify the document length in the OnPreparePrinting member function, where you would normally do so; you must wait until the printer device context has been created based on the dialog box settings. OnBeginPrinting is the first overridable function that gives you access to the CDC object representing the printer device context, so you can set the document length from this function. Note that if the document length is not specified by this time, a scroll bar is not displayed during print preview.

`CView::OnDragEnter`

Called by the framework when the mouse first enters the non-scrolling region of the drop target window.

```
virtual DROPEFFECT OnDragEnter(
    COleDataObject* pDataObject,
    DWORD dwKeyState,
    CPoint point);
```

Parameters

`pDataObject`
Points to the COleDataObject being dragged into the drop area of the view.

`dwKeyState`
Contains the state of the modifier keys. This is a combination of any number of the following: `MK_CONTROL`, `MK_SHIFT`, `MK_ALT`, `MK_LBUTTON`, `MK_MBUTTON`, and `MK_RBUTTON`.

`point`
The current mouse position relative to the client area of the view.

Return Value

A value from the `DROPEFFECT` enumerated type, which indicates the type of drop that would occur if the user dropped the object at this position. The type of drop usually depends on the current key state indicated by *`dwKeyState`*. A standard mapping of keystates to `DROPEFFECT` values is:

- `DROPEFFECT_NONE` The data object cannot be dropped in this window.
- `DROPEFFECT_LINK` for `MK_CONTROL|MK_SHIFT` Creates a linkage between the object and its server.
- `DROPEFFECT_COPY` for `MK_CONTROL` Creates a copy of the dropped object.
- `DROPEFFECT_MOVE` for `MK_ALT` Creates a copy of the dropped object and delete the original object. This is typically the default drop effect, when the view can accept this data object.

For more information, see the MFC Advanced Concepts sample OCLIENT.

Remarks

Default implementation is to do nothing and return `DROPEFFECT_NONE`.

Override this function to prepare for future calls to the OnDragOver member function. Any data required from the data object should be retrieved at this time for later use in the `OnDragOver` member function. The view should also be updated at this time to give the user visual feedback. For more information, see the article OLE drag and drop: Implement a drop target.

`CView::OnDragLeave`

Called by the framework during a drag operation when the mouse is moved out of the valid drop area for that window.

```
virtual void OnDragLeave();
```

Remarks

Override this function if the current view needs to clean up any actions taken during OnDragEnter or OnDragOver calls, such as removing any visual user feedback while the object was dragged and dropped.

`CView::OnDragOver`

Called by the framework during a drag operation when the mouse is moved over the drop target window.

```
virtual DROPEFFECT OnDragOver(
    COleDataObject* pDataObject,
    DWORD dwKeyState,
    CPoint point);
```

Parameters

pDataObject

Points to the COleDataObject being dragged over the drop target.

dwKeyState

Contains the state of the modifier keys. This is a combination of any number of the following: `MK_CONTROL`, `MK_SHIFT`, `MK_ALT`, `MK_LBUTTON`, `MK_MBUTTON`, and `MK_RBUTTON`.

point

The current mouse position relative to the view client area.

Return Value

A value from the `DROPEFFECT` enumerated type, which indicates the type of drop that would occur if the user dropped the object at this position. The type of drop often depends on the current key state as indicated by *dwKeyState*. A standard mapping of keystates to `DROPEFFECT` values is:

- `DROPEFFECT_NONE` The data object cannot be dropped in this window.
- `DROPEFFECT_LINK` for `MK_CONTROL|MK_SHIFT` Creates a linkage between the object and its server.
- `DROPEFFECT_COPY` for `MK_CONTROL` Creates a copy of the dropped object.
- `DROPEFFECT_MOVE` for `MK_ALT` Creates a copy of the dropped object and delete the original object. This is typically the default drop effect, when the view can accept the data object.

For more information, see the MFC Advanced Concepts sample OCLIENT.

Remarks

The default implementation is to do nothing and return `DROPEFFECT_NONE`.

Override this function to give the user visual feedback during the drag operation. Since this function is called continuously, any code contained within it should be optimized as much as possible. For more information, see the article OLE drag and drop: Implement a drop target.

`CView::OnDragScroll`

Called by the framework before calling OnDragEnter or OnDragOver to determine whether the point is in the scrolling region.

```
virtual DROPEFFECT OnDragScroll(
    DWORD dwKeyState,
    CPoint point);
```

Parameters

`dwKeyState`
Contains the state of the modifier keys. This is a combination of any number of the following: `MK_CONTROL`, `MK_SHIFT`, `MK_ALT`, `MK_LBUTTON`, `MK_MBUTTON`, and `MK_RBUTTON`.

`point`
Contains the location of the cursor, in pixels, relative to the screen.

Return Value

A value from the `DROPEFFECT` enumerated type, which indicates the type of drop that would occur if the user dropped the object at this position. The type of drop usually depends on the current key state indicated by *`dwKeyState`*. A standard mapping of keystates to `DROPEFFECT` values is:

- `DROPEFFECT_NONE` The data object cannot be dropped in this window.
- `DROPEFFECT_LINK` for `MK_CONTROL|MK_SHIFT` Creates a linkage between the object and its server.
- `DROPEFFECT_COPY` for `MK_CONTROL` Creates a copy of the dropped object.
- `DROPEFFECT_MOVE` for `MK_ALT` Creates a copy of the dropped object and delete the original object.
- `DROPEFFECT_SCROLL` Indicates that a drag scroll operation is about to occur or is occurring in the target view.

For more information, see the MFC Advanced Concepts sample OCLIENT.

Remarks

Override this function when you want to provide special behavior for this event. The default implementation automatically scrolls windows when the cursor is dragged into the default scroll region inside the border of each window. For more information, see the article OLE drag and drop: Implement a drop target.

`CView::OnDraw`

Called by the framework to render an image of the document.

```
virtual void OnDraw(CDC* pDC) = 0;
```

Parameters

pDC
Points to the device context to be used for rendering an image of the document.

Remarks

The framework calls this function to perform screen display, printing, and print preview, and it passes a different device context in each case. There is no default implementation.

You must override this function to display your view of the document. You can make graphic device interface (GDI) calls using the CDC object pointed to by the *pDC* parameter. You can select GDI resources, such as pens or fonts, into the device context before drawing and then deselect them afterwards. Often your drawing code can be device-independent; that is, it doesn't require information about what type of device is displaying the image.

To optimize drawing, call the RectVisible member function of the device context to find out whether a given rectangle will be drawn. If you need to distinguish between normal screen display and printing, call the IsPrinting member function of the device context.

`CView::OnDrop`

Called by the framework when the user releases a data object over a valid drop target.

```
virtual BOOL OnDrop(
    COleDataObject* pDataObject,
    DROPEFFECT dropEffect,
    CPoint point);
```

Parameters

`pDataObject`

Points to the COleDataObject that is dropped into the drop target.

`dropEffect`

The drop effect that the user has requested.

- `DROPEFFECT_COPY` Creates a copy of the data object being dropped.
- `DROPEFFECT_MOVE` Moves the data object to the current mouse location.
- `DROPEFFECT_LINK` Creates a link between a data object and its server.

`point`

The current mouse position relative to the view client area.

Return Value

Nonzero if the drop was successful; otherwise 0.

Remarks

The default implementation does nothing and returns `FALSE`.

Override this function to implement the effect of an OLE drop into the client area of the view. The data object can be examined via *`pDataObject`* for Clipboard data formats and data dropped at the specified point.

Note

The framework does not call this function if there is an override to OnDropEx in this view class.

CView::OnDropEx

Called by the framework when the user releases a data object over a valid drop target.

```
virtual DROPEFFECT OnDropEx(
    COleDataObject* pDataObject,
    DROPEFFECT dropDefault,
    DROPEFFECT dropList,
    CPoint point);
```

Parameters

pDataObject
Points to the COleDataObject that is dropped into the drop target.

dropDefault
The effect that the user chose for the default drop operation based on the current key state. It may be `DROPEFFECT_NONE`. Drop effects are discussed in the Remarks section.

dropList
A list of the drop effects that the drop source supports. Drop effect values can be combined using the bitwise OR (`|`) operation. Drop effects are discussed in the Remarks section.

point
The current mouse position relative to the view client area.

Return Value

The drop effect that resulted from the drop attempt at the location specified by *point*.

This must be one of the values indicated by *`dropEffectList`*. Drop effects are discussed in the Remarks section.

Remarks

The default implementation is to do nothing and return a dummy value (-1) to indicate that the framework should call the OnDrop handler.

Override this function to implement the effect of an right mouse-button drag and drop. Right mouse-button drag and drop typically displays a menu of choices when the right mouse-button is released.

Your override of `OnDropEx` should query for the right mouse-button. You can call GetKeyState or store the right mouse-button state from your OnDragEnter handler.

- If the right mouse-button is down, your override should display a popup menu which offers the drop effects support by the drop source.
  - Examine *`dropList`* to determine the drop effects supported by the drop source. Enable only these actions on the popup menu.
  - Use SetMenuDefaultItem to set the default action based on *`dropDefault`*.
  - Finally, take the action indicated by the user selection from the popup menu.
- If the right mouse-button is not down, your override should process this as a standard drop request. Use the drop effect specified in *`dropDefault`*. Alternately, your override can return the dummy value (-1) to indicate that `OnDrop` will handle this drop operation.

Use *`pDataObject`* to examine the `COleDataObject` for Clipboard data format and data dropped at the specified point.

Drop effects describe the action associated with a drop operation. See the following list of drop effects:

- `DROPEFFECT_NONE` A drop would not be allowed.
- `DROPEFFECT_COPY` A copy operation would be performed.
- `DROPEFFECT_MOVE` A move operation would be performed.

- `DROPEFFECT_LINK` A link from the dropped data to the original data would be established.
- `DROPEFFECT_SCROLL` Indicates that a drag scroll operation is about to occur or is occurring in the target.

For more information on setting the default menu command, see SetMenuDefaultItem in the Windows SDK and CMenu::GetSafeHmenu in this volume.

`CView::OnEndPrinting`

Called by the framework after a document has been printed or previewed.

```
virtual void OnEndPrinting(
    CDC* pDC,
    CPrintInfo* pInfo);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a CPrintInfo structure that describes the current print job.

Remarks

The default implementation of this function does nothing. Override this function to free any GDI resources you allocated in the OnBeginPrinting member function.

`CView::OnEndPrintPreview`

Called by the framework when the user exits print preview mode.

```
virtual void OnEndPrintPreview(
    CDC* pDC,
    CPrintInfo* pInfo,
    POINT point,
    CPreviewView* pView);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a CPrintInfo structure that describes the current print job.

point

Specifies the point on the page that was last displayed in preview mode.

pView

Points to the view object used for previewing.

Remarks

The default implementation of this function calls the OnEndPrinting member function and restores the main frame window to the state it was in before print preview began. Override this function to perform special processing when preview mode is terminated. For example, if you want to maintain the user's position in the document when switching from preview mode to normal display mode, you can scroll to the position described by the *point* parameter and the `m_nCurPage` member of the `CPrintInfo` structure that the *pInfo* parameter points to.

Always call the base class version of `OnEndPrintPreview` from your override, typically at the end of the function.

`CView::OnInitialUpdate`

Called by the framework after the view is first attached to the document, but before the view is initially displayed.

```
virtual void OnInitialUpdate();
```

Remarks

The default implementation of this function calls the OnUpdate member function with no hint information (that is, using the default values of 0 for the *lHint* parameter and `NULL` for the *pHint* parameter). Override this function to perform any one-time initialization that requires information about the document. For example, if your application has fixed-sized documents, you can use this function to initialize a view's scrolling limits based on the document size. If your application supports variable-sized documents, use OnUpdate to update the scrolling limits every time the document changes.

`CView::OnPrepareDC`

Called by the framework before the OnDraw member function is called for screen display and before the OnPrint member function is called for each page during printing or print preview.

```
virtual void OnPrepareDC(
    CDC* pDC,
    CPrintInfo* pInfo = NULL);
```

Parameters

pDC
Points to the device context to be used for rendering an image of the document.

pInfo
Points to a CPrintInfo structure that describes the current print job if `OnPrepareDC` is being called for printing or print preview; the `m_nCurPage` member specifies the page about to be printed. This parameter is `NULL` if `OnPrepareDC` is being called for screen display.

Remarks

The default implementation of this function does nothing if the function is called for screen display. However, this function is overridden in derived classes, such as CScrollView, to adjust attributes of the device context; consequently, you should always call the base class implementation at the beginning of your override.

If the function is called for printing, the default implementation examines the page information stored in the *pInfo* parameter. If the length of the document has not been specified, `OnPrepareDC` assumes the document to be one page long and stops the print loop after one page has been printed. The function stops the print loop by setting the `m_bContinuePrinting` member of the structure to `FALSE`.

Override `OnPrepareDC` for any of the following reasons:

- To adjust attributes of the device context as needed for the specified page. For example, if you need to set the mapping mode or other characteristics of the device context, do so in this function.
- To perform print-time pagination. Normally you specify the length of the document when printing begins, using the OnPreparePrinting member function. However, if you don't know in advance how long the document is (for example, when printing an undetermined number of records from a database), override `OnPrepareDC` to test for the end of the document while it is being printed. When there is no more of the document to be printed, set the `m_bContinuePrinting` member of the `CPrintInfo` structure to `FALSE`.
- To send escape codes to the printer on a page-by-page basis. To send escape codes from `OnPrepareDC`, call the `Escape` member function of the *pDC* parameter.

Call the base class version of `OnPrepareDC` at the beginning of your override.

Example

C++

```cpp
void CMyView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
   CView::OnPrepareDC(pDC, pInfo);
   // If we are printing, set the mapmode and the window
   // extent properly, then set viewport extent. Use the
   // SetViewportOrg member function in the CDC class to
   // move the viewport origin to the center of the view.

   if (pDC->IsPrinting()) // Is the DC a printer DC.
   {
      CRect rect;
      GetClientRect(&rect);

      pDC->SetMapMode(MM_ISOTROPIC);
      CSize ptOldWinExt = pDC->SetWindowExt(1000, 1000);
      ASSERT(ptOldWinExt.cx != 0 && ptOldWinExt.cy != 0);
      CSize ptOldViewportExt = pDC->SetViewportExt(rect.Width(),
-rect.Height());
      ASSERT(ptOldViewportExt.cx != 0 && ptOldViewportExt.cy != 0);
      CPoint ptOldOrigin = pDC->SetViewportOrg(rect.Width() / 2,
rect.Height() / 2);
   }
}
```

CView::OnPreparePrinting

Called by the framework before a document is printed or previewed.

```
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
```

Parameters

pInfo

Points to a CPrintInfo structure that describes the current print job.

Return Value

Nonzero to begin printing; 0 if the print job has been canceled.

Remarks

The default implementation does nothing.

You must override this function to enable printing and print preview. Call the DoPreparePrinting member function, passing it the *pInfo* parameter, and then return its return value; `DoPreparePrinting` displays the Print dialog box and creates a printer device context. If you want to initialize the Print dialog box with values other than the defaults, assign values to the members of *pInfo*. For example, if you know the length of the document, pass the value to the SetMaxPage member function of *pInfo* before calling `DoPreparePrinting`. This value is displayed in the To: box in the Range portion of the Print dialog box.

`DoPreparePrinting` does not display the Print dialog box for a preview job. If you want to bypass the Print dialog box for a print job, check that the `m_bPreview` member of *pInfo* is `FALSE` and then set it to `TRUE` before passing it to `DoPreparePrinting`; reset it to `FALSE` afterwards.

If you need to perform initializations that require access to the `CDC` object representing the printer device context (for example, if you need to know the page size before specifying the length of the document), override the `OnBeginPrinting` member function.

If you want to set the value of the `m_nNumPreviewPages` or `m_strPageDesc` members of the *pInfo* parameter, do so after calling `DoPreparePrinting`. The `DoPreparePrinting` member function sets `m_nNumPreviewPages` to the value found in the application's .INI file and sets `m_strPageDesc` to its default value.

Example

Override `OnPreparePrinting` and call `DoPreparePrinting` from the override so that the framework will display a Print dialog box and create a printer DC for you.

```C++
BOOL CMyEditView::OnPreparePrinting(CPrintInfo* pInfo)
{
   return CEditView::DoPreparePrinting(pInfo);
}
```

If you know how many pages the document contains, set the maximum page in `OnPreparePrinting` before calling `DoPreparePrinting`. The framework will display the maximum page number in the "to" box of the Print dialog box.

```C++
BOOL CExampleView::OnPreparePrinting(CPrintInfo* pInfo)
{
   //The document has 2 pages.
   pInfo->SetMaxPage(2);
   return CView::DoPreparePrinting(pInfo);
}
```

`CView::OnPrint`

Called by the framework to print or preview a page of the document.

```
virtual void OnPrint(
    CDC* pDC,
    CPrintInfo* pInfo);
```

Parameters

pDC

Points to the printer device context.

pInfo

Points to a `CPrintInfo` structure that describes the current print job.

Remarks

For each page being printed, the framework calls this function immediately after calling the OnPrepareDC member function. The page being printed is specified by the `m_nCurPage` member of the CPrintInfo structure that *pInfo* points to. The default implementation calls the OnDraw member function and passes it the printer device context.

Override this function for any of the following reasons:

- To allow printing of multipage documents. Render only the portion of the document that corresponds to the page currently being printed. If you're using

`OnDraw` to perform the rendering, you can adjust the viewport origin so that only the appropriate portion of the document is printed.

- To make the printed image look different from the screen image (that is, if your application is not WYSIWYG). Instead of passing the printer device context to `OnDraw`, use the device context to render an image using attributes not shown on the screen.

  If you need GDI resources for printing that you don't use for screen display, select them into the device context before drawing and deselect them afterwards. These GDI resources should be allocated in OnBeginPrinting and released in OnEndPrinting.

- To implement headers or footers. You can still use `OnDraw` to do the rendering by restricting the area that it can print on.

Note that the `m_rectDraw` member of the *`pInfo`* parameter describes the printable area of the page in logical units.

Do not call `OnPrepareDC` in your override of `OnPrint`; the framework calls `OnPrepareDC` automatically before calling `OnPrint`.

Example

The following is a skeleton for an overridden `OnPrint` function:

```C++
void CMyView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
   UNREFERENCED_PARAMETER(pInfo);

   // Print headers and/or footers, if desired.
   // Find portion of document corresponding to pInfo->m_nCurPage.
   OnDraw(pDC);
}
```

For another example, see CRichEditView::PrintInsideRect.

`CView::OnScroll`

Called by the framework to determine whether scrolling is possible.

```
virtual BOOL OnScroll(
    UINT nScrollCode,
    UINT nPos,
    BOOL bDoScroll = TRUE);
```

Parameters

`nScrollCode`
A scroll-bar code that indicates the user's scrolling request. This parameter is composed of two parts: a low-order byte, which determines the type of scrolling occurring horizontally, and a high-order byte, which determines the type of scrolling occurring vertically:

- `SB_BOTTOM` Scrolls to bottom.
- `SB_LINEDOWN` Scrolls one line down.
- `SB_LINEUP` Scrolls one line up.
- `SB_PAGEDOWN` Scrolls one page down.
- `SB_PAGEUP` Scrolls one page up.
- `SB_THUMBTRACK` Drags scroll box to specified position. The current position is specified in *`nPos`*.
- `SB_TOP` Scrolls to top.

`nPos`
Contains the current scroll-box position if the scroll-bar code is `SB_THUMBTRACK`; otherwise it is not used. Depending on the initial scroll range, *`nPos`* may be negative and should be cast to an **`int`** if necessary.

`bDoScroll`
Determines whether you should actually do the specified scrolling action. If `TRUE`, then scrolling should take place; if `FALSE`, then scrolling should not occur.

Return Value

If *bDoScroll* is `TRUE` and the view was actually scrolled, then return nonzero; otherwise 0. If *bDoScroll* is `FALSE`, then return the value that you would have returned if *bDoScroll* were `TRUE`, even though you don't actually do the scrolling.

Remarks

In one case this function is called by the framework with *bDoScroll* set to `TRUE` when the view receives a scrollbar message. In this case, you should actually scroll the view. In the other case this function is called with *bDoScroll* set to `FALSE` when an OLE item is initially dragged into the auto-scrolling region of a drop target before scrolling actually takes place. In this case, you should not actually scroll the view.

`CView::OnScrollBy`

Called by the framework when the user views an area beyond the present view of the document, either by dragging an OLE item against the view's current borders or by manipulating the vertical or horizontal scrollbars.

```
virtual BOOL OnScrollBy(
    CSize sizeScroll,
    BOOL bDoScroll = TRUE);
```

Parameters

sizeScroll
Number of pixels scrolled horizontally and vertically.

bDoScroll
Determines whether scrolling of the view occurs. If `TRUE`, then scrolling takes place; if `FALSE`, then scrolling does not occur.

Return Value

Nonzero if the view was able to be scrolled; otherwise 0.

Remarks

In derived classes this method checks to see whether the view is scrollable in the direction the user requested and then updates the new region if necessary. This function is automatically called by CWnd::OnHScroll and CWnd::OnVScroll to perform the actual scrolling request.

The default implementation of this method does not change the view, but if it is not called, the view will not scroll in a `CScrollView`-derived class.

If the document width or height exceeds 32767 pixels, scrolling past 32767 will fail because `OnScrollBy` is called with an invalid *`sizeScroll`* argument.

`CView::OnUpdate`

Called by the framework after the view's document has been modified; this function is called by CDocument::UpdateAllViews and allows the view to update its display to reflect those modifications.

```
virtual void OnUpdate(
    CView* pSender,
    LPARAM lHint,
    CObject* pHint);
```

Parameters

`pSender`
Points to the view that modified the document, or `NULL` if all views are to be updated.

`lHint`
Contains information about the modifications.

`pHint`
Points to an object storing information about the modifications.

Remarks

It is also called by the default implementation of OnInitialUpdate. The default implementation invalidates the entire client area, marking it for painting when the next `WM_PAINT` message is received. Override this function if you want to update only those regions that map to the modified portions of the document. To do this you must pass information about the modifications using the hint parameters.

To use *`lHint`*, define special hint values, typically a bitmask or an enumerated type, and have the document pass one of these values. To use *`pHint`*, derive a hint class from CObject and have the document pass a pointer to a hint object; when overriding `OnUpdate`, use the CObject::IsKindOf member function to determine the run-time type of the hint object.

Typically you should not perform any drawing directly from `OnUpdate`. Instead, determine the rectangle describing, in device coordinates, the area that requires updating; pass this rectangle to CWnd::InvalidateRect. This causes painting to occur the next time a WM_PAINT message is received.

If *`lHint`* is 0 and *`pHint`* is `NULL`, the document has sent a generic update notification. If a view receives a generic update notification, or if it cannot decode the hints, it should invalidate its entire client area.

See also

MFC Sample MDIDOCVW
CWnd Class
Hierarchy Chart
CWnd Class
CFrameWnd Class
CSplitterWnd Class
CDC Class
CDocTemplate Class
CDocument Class