Horizontally Scrollable ListBoxes for Windows 3.x, using C++

Ted Faison

Ted is a writer and developer, specializing in Windows and C++. He has authored two books on C++, and has been programming in C++ since 1988. He is president of Faison Computing, a firm which develops C++ class libraries for DOS and Windows. He can be reached at tedfaison@msn.com

List boxes are among the most commonly used child controls in Windows applications. List boxes are typically used to show lists of files, fonts, or other variable-length lists of textual information. To add a list box to a dialog box, you generally edit a resource file, using programs such as Microsoft's Dialog Editor or Borland's Resource Workshop. Windows handles most of the list box details transparently. For example, if you add strings to a list box, Windows will automatically put a scroll bar on the control when the list box contains more strings than can be displayed in the client area of the list box. Windows handles scroll bar events - such as moving the thumb or clicking the up/down arrows - without any need for user code.

Displaying a list of files in a list box is a somewhat easy task, because filenames have a predefined maximum number of characters. When you create the list box resource, you will generally make the control wide enough to display the longest filename. But what if you use a list box to display strings of varying and unknown length, such as the names of people or the titles of your CD collection ? You could obviously make the list box wide enough to accommodate the widest string you expect, but that would not only look pretty bad, but also waste a great deal of space on the screen. A better approach is to make the list box wide enough to handle the average string, using a horizontal scroll bar to scroll the client area when there are strings that are too long to fit completely. The solution seems immediate: Using a tool such as Resource Workshop, you create a list box, and set Horizontal Scroll Bar property (or enable the WS_HSCROLL style bit). Unfortunately, when you display the list box in your application, you'll find with much surprise that the horizontal scroll bar doesn't appear. Why ? Because Windows doesn't handle all aspects of horizontal scroll bars in list boxes. It is up to the application to supply some additional code to make horizontal scroll bars work correctly.

When you use the WS_VSCROLL style with a list box, Windows keeps track of the number of items added to the control. Knowing the size of the list box's font, the number of items in the list box and the height of the list box's client area, Windows can determine when there are more items that will fit on the control. When this condition is met, Windows adds the vertical scroll bar to the list box.

With horizontal scroll bars, things are a little more complicated, because the width of a string is not only dependent on the font being used, but also on the length of the string. The width of a string measured in pixels is called the *text extent*. Windows doesn't automatically keep track of the extents of the strings inserted into list boxes. That's where your application code comes into play. When you add a string to a list box, you must tell Windows the extent of the string. When you indicate an extent that exceeds the width of the list box's client area, Windows draws the horizontal scroll bar. You tell Windows the extent of a string using the LB_SETHORIZONTALEXTENT message. The WParam parameter indicates the extent of the string.

Keeping track of text extents

When you add more than one string to a list box, all Windows needs to know is the extent of the widest string, which means that somehow your application has to keep track of the extents of all the strings in a list box. Using ordinary C code, the management of text extents takes a significant amount of effort. Using C++ containers and a Windows application framework such as OWL (ObjectWindows Library), the job is considerably simpler.

The first step is to create a container class for the text extents. In C++, containers can hold any kind of objects, include ints, chars, pointers and class objects. A simple container is not quite sufficient to handle our text extents, because we need to know the extent of the widest string in the list box. What we need is a sorted container -- one that stores the text extents in ascending order. Each time a string is added to the list box, its extent is computed and added to the container. With a sorted container, the greatest extent will always be the last item in the container.

The implementation of a sorted container restricts the type of objects that can be put into the container. When a new item is added, the container needs to compare the new item with those already in the container, in order to locate the correct insertion point. The comparison is achieved by calling the member functions **operator**< and **operator**== for the item inserted. Because member functions are invoked, only class objects can be used in sorted containers. Using Borland containers, you have a choice on the type of container library to use. For this article, I'll use the template container classes defined in the BIDS (Borland International Data Structure) container class library. With BIDS containers, objects inserted into a sorted container must have the following member functions

```
Default constructor
Copy constructor
Operator ==
Operator <
isSortable()
```

I wrote a small class called Extent to handle the text extents. The class is shown in Listing 1.

```
class Extent {

  int value;

public:

  Extent(int i) {value = i;}
  Extent() {value = 0;}
  Extent(Extent& i) {value = (int) i;}
  int operator==(Extent& i) const {return value == (int) i;}
  int operator<(Extent& i) const {return value < (int) i;}
  operator int() {return value;}
  virtual int isSortable() {return 1;}
};
```

Listing 1 - A class to handle text extents.

Having a class for the text extents, we can create a template-based container to manage the extents. Using BIDS containers, the name of the class we need is BI_SArrayAsVector. The BI letters stand for (what else?) Borland International. The S indicates a sorted array. To create a container called MyContainer to hold Extent objects, the notation:

```
  BI_SArrayAsVector<Extent>  MyContainer;
```

is used. Note that any time you define a template-based variable, the C++ compiler creates a completely new class, based on the template variable.

Having created a container for text extents, the next step is to use the container to store, sort and manage the extents. Before we can add an extent to the container, we need to find what the extent for a string is. I wrote a small function called TextExtent(LPSTR) to do the job. The function computes the extent of a string using the font selected into the list box. Function TextExtent(LPSTR) is coded like this:

```
int THorizontalListBox::TextExtent(LPSTR AString)
{
  int extent;

  // select the ListBox into the device context
  HDC hdc = GetDC(HWindow);
  HFONT hfont = (HFONT) SendMessage(HWindow, WM_GETFONT, 0, 0);

  if (hfont) {
    // non-system font being used: select it into the
    // ListBox's device context before calling GetTextExtent
    HGDIOBJ oldObject = SelectObject(hdc, hfont);

    // find the text extent of the string
    extent = GetTextExtent(hdc, AString, _fstrlen(AString) );

    // release resources used
    SelectObject(hdc, oldObject);
    ReleaseDC(HWindow, hdc);
  }
  else {
    // system font in use: no font selection necessary,
    // because GetTextExtent will use the system font
    // by default
    extent = GetTextExtent(hdc, AString, _fstrlen(AString) );
    ReleaseDC(HWindow, hdc);
  }
  return extent;
}
```

Listing 2 - A function that computes the extent of a string, based on the font selected.

A WM_GETFONT message is sent to the list box to retrieve the handle of the font used in the control. If the system font is being used (which is often the case), a NULL handle is returned. If a non-system font is being used, it must be selected into the list box's device context before calling the Windows API function GetTextExtent.

Having seen how to compute a string's extent, we can add the extent to the container with the code:

```
  int length = TextExtent(AString);
  Extent extent = *new Extent(length);
  textExtents.add(extent);
```

where the variable *textExtents* is the extent container, and *AString* is a pointer to a null-terminated array of characters. The *extent* object inserted into the container is created with the global **new** operator. You have to be a little careful with objects that are put into containers, because by default containers own the

items they contain. What this means is that when the container goes out of scope, it will try to delete all the items that are still in it. If you had put a stack-based or global object in the container, your application would most likely crash, since the **delete** operator may be called only for objects created by the **new** operator. To place stack-based or global objects into a container, you must tell the container that it doesn't own the elements in it. You do so by calling the container's member function **ownsElements()**, passing the value 0. Once ownership is disabled, the container will never try to delete any of the items in it.

Creating an OWL custom control

One way to handle horizontal scroll bars is to use a standard list box, and make the parent window do all the work, managing the text extent container, sending WM_SETHORIZONTALEXTENT messages to the list box, and so on. A better, more object-oriented approach, is to create a new control that does all the housekeeping by itself, without disturbing the parent. With OWL, building a custom control of this type is not difficult. OWL already defines the class TListBox to act as an object-oriented stand-in for regular Windows list boxes. All you have to do is derive a new class from TListBox and add the necessary support for horizontal scroll bars.

I created a small class called THorizontalListBox to encapsulate all the details described. The class declaration is shown in listing 3.

```
class THorizontalListBox: public TListBox {

  class Extent {

    int value;

  public:

    Extent(int i) {value = i;}
    Extent() {value = 0;}
    Extent(Extent& i) {value = (int) i;}
    int operator==(Extent& i) const {return value == (int) i;}
    int operator<(Extent& i) const {return value < (int) i;}
    operator int() {return value;}
    virtual int isSortable() {return 1;}
  };

  typedef BI_SArrayAsVector<Extent> Extents;

  Extents textExtents;

  int TextExtent(LPSTR);
  void UpdateHorizontalExtent();

public:

  THorizontalListBox(PTWindowsObject, int, PTModule = NULL);

  int AddString(LPSTR);
  int InsertString(LPSTR, int);
  int DeleteString(int);
  void ClearList();
  void InsertExtent(LPSTR);
  virtual WORD Transfer(void*, WORD);
```

```
};
```

Listing 3 - The declaration of class THorizontalListBox.

Because class Extent is designed to be used exclusively inside THorizontalListBox, it is declared as a nested class. Class Extent is therefore in scope only inside class THorizontalListBox. The template-based container is typedef'd, to improve code readability. Using the type Extents is a lot easier than having to use BI_SArrayAsVector<Extent>.

The data member textExtents is the actual container for the text extents. The base class functions AddString, InsertString, DeleteString and ClearList all are overridden, because every time a string is added or removed from the list box, the extents container needs to be updated, and the list box's horizontal extent kept up to date. For example, the function THorizontalListBox::AddString(LPSTR) looks like this:

```
int THorizontalListBox::AddString(LPSTR AString)
{
  InsertExtent(AString);
  return TListBox::AddString(AString);
}
```

The function THorizontalListBox::InsertExtent(LPSTR) computes a string's extent, adds it to the extent container, and updates the list box's horizontal extent. The base class function TListBox::AddString(LPSTR) is then called to actually add the string to the list box.

The function THorizontalListBox::DeleteString is similar to AddString, except that it needs to locate the extent of the string being removed from the list box, delete the extent object, then update the list box's horizontal extent. The code looks like this:

```
int THorizontalListBox::DeleteString(int Index)
{
  // find the text extent of the string to be deleted
  char string [256];
  GetString(string, Index);

  // remove the extent from the container
  Extent extent = Extent(TextExtent(string) );
  for (int i = 0; i < textExtents.getItemsInContainer(); i++) {
    if (extent == textExtents [i]) {
      textExtents.detach(extent, TShouldDelete::Delete);
      break;
    }
  }

  // update the ListBox horizontal extent
  UpdateHorizontalExtent();

  return TListBox::DeleteString(Index);
}
```

Listing 4 - Deleting a string from THorizontalListBox.

A short loop searches the extent container for the extent of the string being removed from the list box. The statement:

```
      textExtents.detach(extent, TShouldDelete::Delete);
```

removes the extent object from the container and deletes the object itself. UpdateHorizontalExtent() sets the extent of the list box using the greatest extent still stored in the extent container. The base class function TListBox::DeleteString does the actual work of removing a string from the list box.

Supporting the Transfer Buffer

The only tricky feature of class THorizontalListBox is the way it handles data that is inserted into the list box directly from a transfer buffer. Using OWL, child controls can be setup to operate with a special buffer called a *transfer buffer*. When the child control is created, it reads its initial data from the transfer buffer. When the child control is destroyed, its data can be saved into the transfer buffer. Using a transfer buffer drastically simplifies the task of reading and writing data to and from child controls.

When strings are put into a THorizontalListBox object from a transfer buffer, OWL calls the function WORD THorizontalListBox::Transfer(void*, WORD) to do the job, passing the value TF_SETDATA as the second parameter. Class THorizontalListBox needs to invoke the base class to successfully copy the string data from the transfer buffer into the control. Then the extent container must be updated for each string in the control. The function THorizontalListBox::Transfer(void*, WORD) looks like this:

```
static void AddTextExtent(Object& AString, void* P)
{
  THorizontalListBox* listBox = (THorizontalListBox*) P;
  LPSTR string = (char*)(const char*)(RString)AString;
  if (AString != NOOBJECT)
    listBox->InsertExtent(string);
}

WORD THorizontalListBox::Transfer(void* DataPtr, WORD TransferFlag)
{
  WORD value = TListBox::Transfer(DataPtr, TransferFlag);

  if (TransferFlag == TF_SETDATA) {
    // for each string in the transfer buffer,
    // add its extent to the extent container
    TListBoxData* ListBoxData = *(PTListBoxData*) DataPtr;
    ListBoxData->Strings->forEach(AddTextExtent, this);
  }
  return value;
}
```

Listing 5 - Transfering data into the list box from a transfer buffer.

The first parameter passed to Transfer(void*, WORD) is a pointer to a pointer to a TListBoxData object. This object contains an Array data member called Strings, which holds the strings to be copied into the list box.
After calling the base class to copy the strings, Transfer(void*, WORD) uses the forEach iterator function to iterate over the text extents stored in the container. For each extent found, the function AddTextExtent is called to update the container object.

A Short Example

I wrote a short OWL application -- called HORSCROL -- to demonstrate the use of class THorizontalListBox. The program displays a dialog box that initially looks like this:

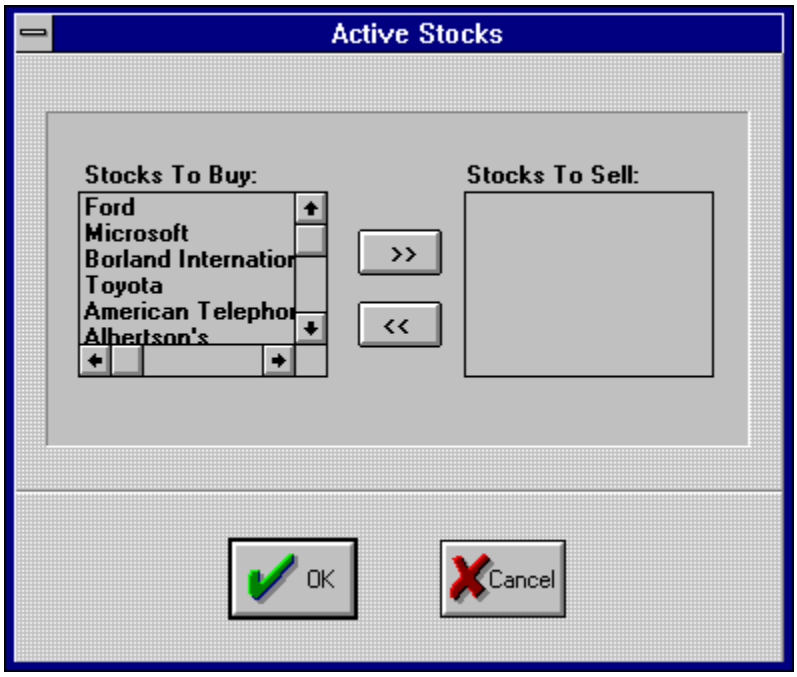

Figure 1 - A dialog box using THorizontalListBox objects.

Using the >> and << buttons, you can move selected entries from one list box to the other, and see the way the horizontal scroll bar is affected. Windows displays the scroll bar only if a list box contains strings that are wider than the list box. For example, adding the string Ford to the right list box, the dialog box looks like this:

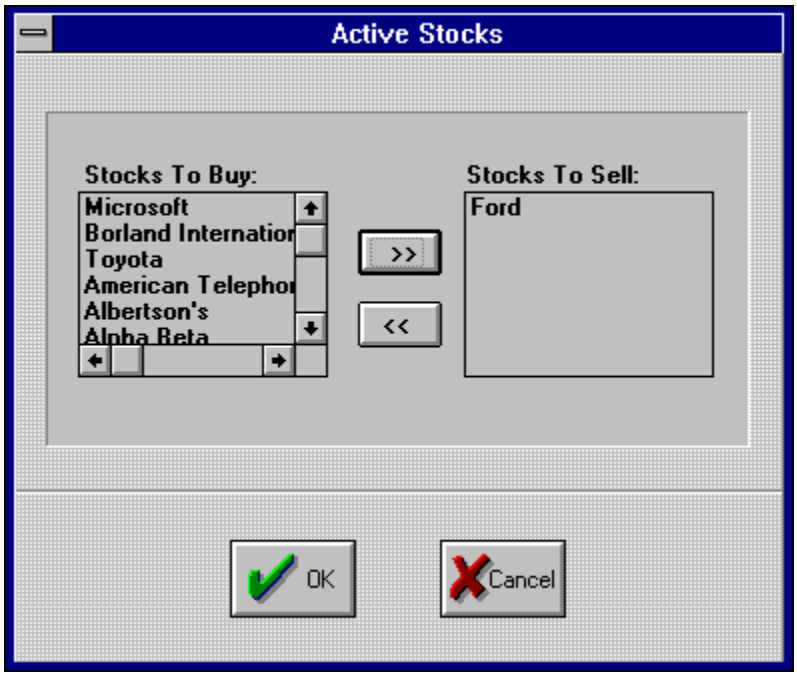

Figure 2 - Adding only a short string to the list box on the right.

To get Windows to display a horizontal scroll bar, a list box must be created with both the WS_VSCROLL and WS_HSCROLL styles. If you omit the WS_VSCROLL style, Windows adds it automatically. The physical act of displaying a horizontal scroll bar is independent from the display of a vertical scroll bar. For example, adding a long string to the right list box causes a horizontal scroll bar, but no vertical one to appear, as shown in the figure 3.

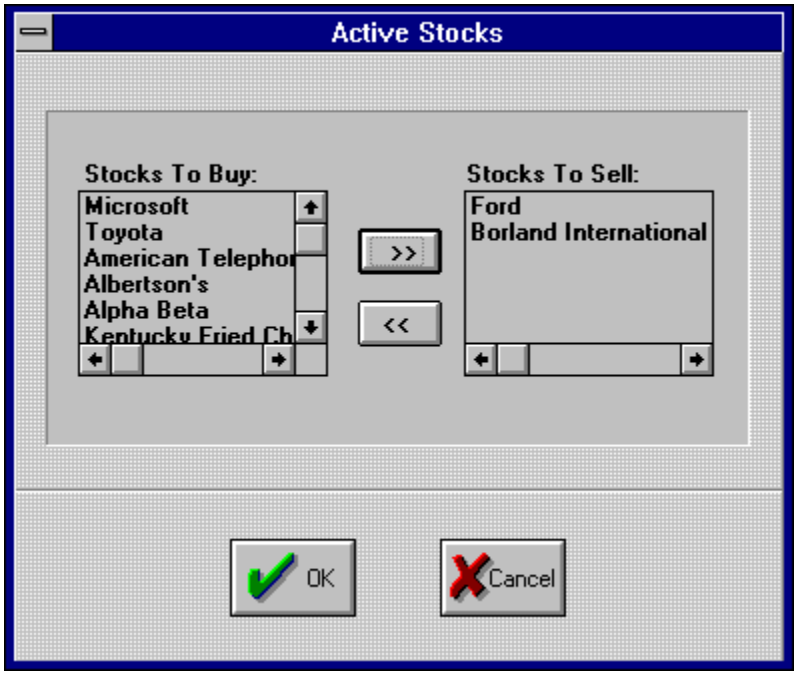

Figure 3 - A list box with both scroll bars, and a list box with only a horizontal scroll bar.

The code for HORSCROL is contained in 3 separate files, with associated header files:

- HLBOX.CPP Custom List Box Control
- DLGSTOCK.CPP Dialog box code
- HORSCROLL.CPP OWL application code

Each file is shown in the following listings.

```
#ifndef __HLBOX_HPP
#define __HLBOX_HPP

#include <arrays.h>
#include <owl.h>
#include <listbox.h>


class THorizontalListBox: public TListBox {

  class Extent {

    int value;

  public:

    Extent(int i) {value = i;}
    Extent() {value = 0;}
    Extent(Extent& i) {value = (int) i;}
    int operator==(Extent& i) const {return value == (int) i;}
    int operator<(Extent& i) const {return value < (int) i;}
    operator int() {return value;}
    virtual int isSortable() {return 1;}
  };

  typedef BI_SArrayAsVector<Extent> Extents;

  Extents textExtents;

  int TextExtent(LPSTR);
  void UpdateHorizontalExtent();

public:

  THorizontalListBox(PTWindowsObject, int, PTModule = NULL);

  int AddString(LPSTR);
  int InsertString(LPSTR, int);
  int DeleteString(int);
  void ClearList();
  void InsertExtent(LPSTR);
  virtual WORD Transfer(void*, WORD);
};

#endif
```

Listing 6 - The header file for class THorizontalListBox.

```
#include "hlbox.hpp"

THorizontalListBox::THorizontalListBox(TWindowsObject* AParent,
                                       int id,
                                       TModule* module)
                  : TListBox(AParent, id, module),
                    textExtents(20, 0, 20)
{
}

static void AddTextExtent(Object& AString, void* P)
{
  THorizontalListBox* listBox = (THorizontalListBox*) P;
  LPSTR string = (char*)(const char*)(RString)AString;
  if (AString != NOOBJECT)
    listBox->InsertExtent(string);
}

WORD THorizontalListBox::Transfer(void* DataPtr, WORD TransferFlag)
{
  WORD value = TListBox::Transfer(DataPtr, TransferFlag);

  if (TransferFlag == TF_SETDATA) {
    // for each string in the transfer buffer,
    // add its extent to the extent container
    TListBoxData* ListBoxData = *(PTListBoxData*) DataPtr;
    ListBoxData->Strings->forEach(AddTextExtent, this);
  }
  return value;
}

int THorizontalListBox::AddString(LPSTR AString)
{
  InsertExtent(AString);
  return TListBox::AddString(AString);
}

int THorizontalListBox::InsertString(LPSTR AString, int Index)
{
  InsertExtent(AString);
  return TListBox::InsertString(AString, Index);
}

void THorizontalListBox::InsertExtent(LPSTR AString)
{
  // store the extent of each string in sorted
  // order in a container
  int length = TextExtent(AString);
  Extent extent = *new Extent(length);
  textExtents.add(extent);

  // update the ListBox horizontal extent
  UpdateHorizontalExtent();
}

int THorizontalListBox::DeleteString(int Index)
{
  // find the text extent of the string to be deleted
  char string [256];
  GetString(string, Index);
```

```
  // remove the extent from the container
  Extent extent = Extent(TextExtent(string) );
  for (int i = 0; i < textExtents.getItemsInContainer(); i++) {
    if (extent == textExtents [i]) {
      textExtents.detach(extent, TShouldDelete::Delete);
      break;
    }
  }

  // update the ListBox horizontal extent
  UpdateHorizontalExtent();

  return TListBox::DeleteString(Index);
}

void THorizontalListBox::ClearList()
{
  // delete all the text extents in the container
  textExtents.flush();

  // update the ListBox horizontal extent
  UpdateHorizontalExtent();

  // Call DeleteString, to force Windows 3.0 to remove
  // the horizontal scrollbar. Windows 3.1 doesn't need
  // this call
  DeleteString(0);

  // clear out the remaining strings in the ListBox
  TListBox::ClearList();
}

// find the extent of a ListBox string
int THorizontalListBox::TextExtent(LPSTR AString)
{
  int extent;

  // select the ListBox into the device context
  HDC hdc = GetDC(HWindow);
  HFONT hfont = (HFONT) SendMessage(HWindow, WM_GETFONT, 0, 0);

  if (hfont) {
    // non-system font being used: select it into the
    // ListBox's device context before calling GetTextExtent
    HGDIOBJ oldObject = SelectObject(hdc, hfont);

    // find the text extent of the string
    extent = GetTextExtent(hdc, AString, _fstrlen(AString) );

    // release resources used
    SelectObject(hdc, oldObject);
    ReleaseDC(HWindow, hdc);
  }

  else {
    // system font in use: no font selection necessary,
    // because GetTextExtent will use the system font
    // by default
    extent = GetTextExtent(hdc, AString, _fstrlen(AString) );
```

```
    ReleaseDC(HWindow, hdc);
  }

  return extent;
}

void THorizontalListBox::UpdateHorizontalExtent()
{
  int greatestExtent;

  // find the extent of the longest string in the
  // ListBox, and set the horizontal extent accordingly
  int lastElement = textExtents.getItemsInContainer() - 1;

  if (lastElement < 0)
    // no more strings in the ListBox
    greatestExtent = 0;
  else
    greatestExtent = textExtents [lastElement];

  // add a small amount of space, so that when the
  // ListBox is completely scrolled to the right,
  // the last character is completely visible
  HDC hdc = GetDC(HWindow);
  greatestExtent += GetTextExtent(hdc, "X", 1);
  ReleaseDC(HWindow, hdc);

  // if the longest string fits completely in the ListBox,
  // then scroll the box completely to the left, so
  // Windows will hide the scrollbar
  RECT rect;
  GetClientRect(HWindow, (LPRECT) &rect);
  int listWidth = rect.right - rect.left;
  if (listWidth >= greatestExtent)
    SendMessage(HWindow, WM_HSCROLL, SB_TOP, 0);

  // set the extent
  SendMessage(HWindow, LB_SETHORIZONTALEXTENT, greatestExtent, 0);
}
```

Listing 7 - The source code for class THorizontalListBox.

```
#ifndef __DLGSTOCKS_HPP
#define __DLGSTOCKS_HPP

#include <owl.h>

#include "hlbox.hpp"

// dialog box child control IDs
const int
  IDC_ADDSELL = 101,
  IDC_ADDBUY  = 102,
  IDC_TOBUY   = 201,
  IDC_TOSELL  = 202;

// initializer class for the Stocks dialog box
class TDialogStocksInitializer {
```

```
public:

  struct {
    PTListBoxData toBuy;
    PTListBoxData toSell;
  } Data;

   TDialogStocksInitializer();
   ~TDialogStocksInitializer();
};

class TDialogStocks: public TDialog {

  THorizontalListBox* toBuy;
  THorizontalListBox* toSell;

  void MoveString(THorizontalListBox*,
             THorizontalListBox*);

public:

  TDialogStocks(PTWindowsObject);
  virtual void AddBuy(RTMessage)  = [ID_FIRST + IDC_ADDBUY];
  virtual void AddSell(RTMessage) = [ID_FIRST + IDC_ADDSELL];
};

#endif
```

Listing 8 - The header file for class TDialogStocks.

```
#include "dlgstock.hpp"

// initializer for the dialog box controls
static TDialogStocksInitializer buffer;

TDialogStocksInitializer::TDialogStocksInitializer()
{
  Data.toBuy = new TListBoxData;
  Data.toBuy->AddString("Ford");
  Data.toBuy->AddString("Microsoft");
  Data.toBuy->AddString("Borland International");
  Data.toBuy->AddString("Toyota");
  Data.toBuy->AddString("American Telephone and Telegraph
Incorporated");
  Data.toBuy->AddString("Albertson's");
  Data.toBuy->AddString("Alpha Beta");
  Data.toBuy->AddString("Kentucky Fried Chicken");
  Data.toBuy->AddString("MacDonald's");
  Data.toBuy->AddString("Pizza Hut");
  Data.toBuy->AddString("Target Stores");
  Data.toSell = new TListBoxData;
}

TDialogStocksInitializer::~TDialogStocksInitializer()
{
  delete Data.toBuy;
  delete Data.toSell;
}
```

```
TDialogStocks::TDialogStocks(PTWindowsObject AParent)
          : TDialog(AParent, "DIALOG_STOCKS")
{
  toBuy  = new THorizontalListBox(this, IDC_TOBUY);
  toSell = new THorizontalListBox(this, IDC_TOSELL);

  TransferBuffer = &buffer.Data;
}

void TDialogStocks::MoveString(THorizontalListBox* from,
                         THorizontalListBox* to)
{
  int selection = from->GetSelIndex();
  if (selection < 0)
    // no selections
    return;

  char string [80];
  const int SEARCH_ENTIRE_LIST = -1;
  from->GetString(string, selection);
  if (to->FindExactString(string, SEARCH_ENTIRE_LIST) >= 0)
    // string already in destination list
    return;

  // move the string between the list boxes
  from->DeleteString(selection);
  to->AddString(string);
}

void TDialogStocks::AddBuy(RTMessage)
{
  MoveString(toSell, toBuy);
}

void TDialogStocks::AddSell(RTMessage)
{
  MoveString(toBuy, toSell);
}
```

Listing 9 - The source code for class TDialogStocks.

```
// this application demonstrates how to implement horizontal
// scrollbars ListBoxes

#include <owl.h>

#include "dlgstock.hpp"

// menu commands
const int
  IDM_STOCKS    = 101,
  IDM_HELPABOUT = 201;

// miscellaneous
LPSTR APPLICATION_NAME = "Using Horizontally Scrollable List Boxes";

class TDialogAbout: public TDialog {
```

```
public:

  TDialogAbout(PTWindowsObject AParent)
    : TDialog(AParent, "DIALOG_ABOUT") {}
};

class TWindowMain : public TWindow {

public:

  TWindowMain();

protected:

  virtual void Stocks(RTMessage)
    = [CM_FIRST + IDM_STOCKS];

  virtual void HelpAbout(RTMessage)
    = [CM_FIRST + IDM_HELPABOUT];
};

TWindowMain::TWindowMain()
         : TWindow(NULL, APPLICATION_NAME)
{
  AssignMenu("MENU_MAIN");
}

void TWindowMain::Stocks(RTMessage)
{
  GetModule()->ExecDialog(new TDialogStocks(this) );
}

void TWindowMain::HelpAbout(RTMessage)
{
  GetModule()->ExecDialog(new TDialogAbout(this) );
}

class TUserApplication: public TApplication {

public:

  TUserApplication(LPSTR AName,
                   HINSTANCE AnInstance,
               HINSTANCE APrevInstance,
               LPSTR ACmdLine,
               int ACmdShow)

    :  TApplication(AName, AnInstance,
                APrevInstance,
                ACmdLine, ACmdShow) {}
  virtual void InitMainWindow();
};


void TUserApplication::InitMainWindow()
{
  MainWindow = new TWindowMain();
}

int PASCAL WinMain(HINSTANCE AnInstance, HINSTANCE APrevInstance,
```

```
                LPSTR ACmdLine, int ACmdShow)
{
  TUserApplication Application(APPLICATION_NAME,
                          AnInstance,
                          APrevInstance,
                          ACmdLine,
                          ACmdShow);
  Application.nCmdShow = SW_SHOWMAXIMIZED;
  Application.Run();
  return Application.Status;
}
```

Listing 10 - The main application code for HORSCROL.

Conclusion

Managing horizontal scroll bars in list boxes is not something application programs should be required to do. Windows should bear the responsibility for providing full support, just as it does with vertical scroll bars. Complete support is likely to appear in the next release of Windows, but in the meantime developers are stuck rolling their own scroll bars.

Recognizing the difficulty of implementing horizontally scrollable list boxes in C, Microsoft has written a technical bulletin to provide some support for developers. The bulletin was written by Kraig Brockschmidt and Kyle Marsh, and is available both on CompuServe, in the Developer's Network Forum, and on the Developer's Network CD. The bulletin's filename is LISTHORZ, and includes DLL code to support horizontal list boxes for C application. See the references at the end of the article for additional information

Using OWL code and C++ containers makes the job not only much easier, but also much more elegant. There are no DLLs to load, and objects of class THorizontalListBox are direct replacements for TListBox objects. The neat thing about using objects is that users don't have to know much (or even anything!) about how the objects are implemented in order to use them. Class THorizontalListBox doesn't have any extra functions for users to call beyond those that are needed by class TListBox.

The way class THorizontalListBox is implemented makes it suitable only for static linkage to application code. To compile the class into a DLL, use the keyword _EXPORT in the class declarations for THorizontalListBox and Extent. Vive les objects!

References:

Considerations for Horizontal Scroll Bars in List Boxes, Microsoft Developer's Network CD, Kraig Brockschmidt and Kyle Marsh, March 1992