Adding a lookup control to the dictionary: Searching Pinyin

devblogs.microsoft.com/oldnewthing/20050812-10

August 12, 2005

Raymond Chen

Finally we start searching. For now, the search algorithm is going to be very simple: The string you type into the edit control will be treated as the start of a Pinyin word or phrase. We'll make it fancier later.

Here is where a lot of the groundwork (some of which I called out explicitly and some of which I slipped in without calling attention to it) starts to pay off.

Up until now, the items in the listview came directly from the dictionary. Of course, when a word is being looked up, we want to reduce the list to those that match the word or phrase being searched for. We will introduce a new member `m_vMatch` which is a vector of pointers to the items we actually want to display.

```
class RootWindow : public Window
{
 …
 // const DictionaryEntry& Item(int i) { return m_dict.Item(i); }
 // int Length() { return m_dict.Length(); }
 const DictionaryEntry& Item(int i) { return *m_vMatch[i]; }
 int Length() { return m_vMatch.size(); }
 …
 void OnCommand(UINT id, UINT cmd);
 void Refilter();
 …
private:
 …
 vector<const DictionaryEntry*> m_vMatch;
};
```

By tweaking our `Item` and `Length` member functions, we can now render out of the list of matches instead of out of the entire dictionary.

```
LRESULT RootWindow::OnCreate()
{
 …
 // ListView_SetItemCount(m_hwndLV, Length());
 …
 m_hwndLastFocus = m_hwndEdit;
 m_vMatch.reserve(m_dict.Length());
 Refilter();


 return 0;
}
```

Since the list of matches is at most the number of words in the dictionary, we can reserve that size up front and avoid needless reallocations. Once we've done that, we call our new `Refilter` method to compute the matches (which populates the listview). It is `Refilter` that will do the `ListView_SetItemCount`, so there's no point in us doing it here.

```
void RootWindow::OnCommand(UINT id, UINT cmd)
{
 switch (id) {
 case IDC_EDIT:
  switch (cmd) {
  case EN_CHANGE:
   Refilter();
  }
  break;
 }
}


  // add to RootWindow::HandleMessage()
  case WM_COMMAND:
   OnCommand(GET_WM_COMMAND_ID(wParam, lParam),
             GET_WM_COMMAND_CMD(wParam, lParam));
   break;
```

We also rebuild the list of matches if the user makes a change to the edit control. This means that there is no need for a "Search" button. The listview auto-filters as you type.

```
void RootWindow::Refilter()
{
 WCHAR szBuf[256];
 DWORD cchBuf = GetWindowText(m_hwndEdit, szBuf, 256);
 m_vMatch.clear();
 for (int i = 0; i < m_dict.Length(); i++) {
  const DictionaryEntry& de = m_dict.Item(i);
  if (StrCmpNIW(de.m_pszPinyin, szBuf, cchBuf) == 0) {
   m_vMatch.push_back(&de);
  }
 }
 ListView_SetItemCount(m_hwndLV, Length());
 ListView_SetItemState(m_hwndLV, -1, 0, LVIS_SELECTED);
 InvalidateRect(m_hwndLV, NULL, FALSE);
}
```

Building the list of matches is rather simple and anticlimactic. We get the string the user typed into the edit control and walk through all the words in the dictionary, seeing if the Pinyin begins with the user's typing. If so, then we add it to the match vector.

Once the match list is built up, we tell the listview how many we found, clear the selection (so that the selection doesn't appear to move around from one word to another as items are filtered in or out), and invalidate the client rectangle to trigger a repaint.

That's all there is to it. If you run this program and start typing into the edit control, you'll see the list of words in the listview grow and shrink as you type.

That's all for this month. Next month, we'll work on expanding the scope of the search.

Raymond Chen

Follow