Loading the dictionary, part 1: Starting point

devblogs.microsoft.com/oldnewthing/20050510-55

May 10, 2005

Raymond Chen

The first thing we'll need to do in our little dictionary program is to load the dictionary into memory. The format of the dictionary file is as a plain text file, each line of which is of the form

```
Chinese [pinyin] /English 1/English 2/.../
舉例 [ju3 li4] /to give an example/
```

Since it was the Big5 dictionary we downloaded, the Chinese characters are in Big5 format, known to Windows as code page 950. Our program will be Unicode, so we'll have to convert it as we load the dictionary. Yes, I could've used the Unicode version of the dictionary, but it so happens that when I set out to write this program, there was no Unicode version available. Fortunately, this oversight opened up the opportunity to illustrate some other programming decisions and techniques.

The first stage in our series of exercises will be loading the dictionary into memory.

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <string>
#include <fstream>
#include <iostream> // for cin/cout
#include <vector>
using std::string;
using std::wstring;
using std::vector;
struct DictionaryEntry
{
 bool Parse(const wstring& line);
 wstring trad;
 wstring simp;
 wstring pinyin;
 wstring english;
};
bool DictionaryEntry::Parse(const wstring& line)
{
    wstring::size_type start = 0;
    wstring::size_type end = line.find(L' ', start);
    if (end == wstring::npos) return false;
    trad.assign(line, start, end);
    start = line.find(L'[', end);
    if (start == wstring::npos) return false;
    end = line.find(L']', ++start);
    if (end == wstring::npos) return false;
    pinyin.assign(line, start, end - start);
    start = line.find(L'/', end);
    if (start == wstring::npos) return false;
    start++;
    end = line.rfind(L'/');
    if (end == wstring::npos) return false;
    if (end <= start) return false;
    english.assign(line, start, end-start);
    return true;
}
class Dictionary
{
public:
 Dictionary();
 int Length() { return v.size(); }
 const DictionaryEntry& Item(int i) { return v[i]; }
private:
 vector<DictionaryEntry> v;
};
Dictionary::Dictionary()
{
 std::wifstream src;
 src.imbue(std::locale(".950"));
 src.open("cedict.b5");
 wstring s;
 while (getline(src, s)) {
  if (s.length() > 0 && s[0] != L'#') {
   DictionaryEntry de;
   if (de.Parse(s)) {
    v.push_back(de);
   }
  }
 }
}
int __cdecl main(int argc, const char* argv[])
{
 DWORD dw = GetTickCount();
 {
  Dictionary dict;
  std::cout << dict.Length() << std::endl;
  std::cout << GetTickCount() - dw << std::endl;
 }
 std::cout << GetTickCount() - dw << std::endl;
 return 0;
}
```

Our dictionary is just a list of words with their English definitions. The Chinese words are written in three forms (traditional Chinese, simplified Chinese, and Pinyin romanization). For those who are curious, there are two writing systems for the Mandarin Chinese language and two phonetic systems. Which one a particular Mandarin-speaking population follows depends on whether they fell under the influence of China's language reform of 1956. Traditional Chinese characters and the Bopomo phonetic system (also called Bopomofo) are used on Taiwan; simplified Chinese characters and the Pinyin system are used in China. Converting Pinyin to Bopomo isn't interesting, so I've removed that part from the program I'm presenting here.

(The schism in the spelling of the English language follows a similar pattern. Under the leadership of Noah Webster, the United States underwent its own spelling reform, but countries which were under the influence of the British crown retained the traditional spellings. Spelling reform continues in other languages even today, and the subject is almost always highly contentious, with traditionalists and reformists pitted against each other in a battle over a language's—and by proxy, a culture's—identity.)

The program itself is fairly straightforward. It creates a Unicode file stream `wifstream` and "imbues" it with code page 950 (Big5). This instructs the runtime to interpret the bytes of the file interpreted in the specified code page. We read strings out of the file, ignore the comments, and parse the rest, appending them to our `vector` of dictionary entries.

Parsing the line consists of finding the spaces, brackets, and slashes, and splitting the line into the traditional Chinese, Pinyin, and English components. (We'll deal with simplified Chinese later.)

When I run this program on my machine, the dictionary loads in 2080ms (or 2140ms if you include the time to run the destructor). This is an unacceptably long startup time, so the first order of business is to make startup faster. That will be the focus of this stage.

Notice that as a sanity check, I print the total number of words in the dictionary. The number should match the number of lines in the `cedict.b5` file (minus the one comment line). If not, then I know that something went wrong. **This is an important sanity check**: You might make a performance optimization that looks great when you run it past a stopwatch, only to discover that your “optimization” actually introduced a bug. For example, one of my attempted optimizations of this program resulted in a phenomenal tenfold speedup, but only because of a bug that caused it to think it was finished when it had in reality processed only 10% of the dictionary!

As my colleague Rico Mariani is fond of saying, “It’s easy to make it fast if it doesn’t have to work!”

Raymond Chen

Follow