

## Re: passing a string to a dll

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-09/msg00970.html>

---

- *From:* Joseph M. Newcomer <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)>
  - *Date:* Thu, 20 Sep 2007 02:52:23 -0400
- 

First, you can put a STRINGTABLE in a DLL. That's how I built a compressed database.

What I did was, with an external tool, I created a STRINGTABLE that was sorted, and the table was created from 1 to n for as many strings I had. For example

```
1 Abalone
2 Broccoli
3 Carrots
4 Dhal
5 Eggs
6. French Fries
7 Guacamole
....
25 Yams
26 Zest of lemon
```

So to do the bsearch, I would make my first estimate, 13. LoadString. If < estimate downward, if > estimate upward, etc., but there is a LoadString to load the string. That's expensive.

To speed this up, I did a first-letter index

```
1 FASTINDEX
BEGIN
1, 'A',
2200, 'B',
6130, 'C',
....
38203, 'Z',
39882, 0,
END
```

so this was set up on a letter by letter basis (the LoadStrings were expensive). So I first did a bsearch on this array, which I read in just once at the start of the program. I had about 40,000 records. So once I had the range narrowed down, I knew the region within which I would be running, e.g., something that started with B would be between entry 2200 and 6129. This meant that my  $n \log n$  search had a vastly smaller  $n$  to work on. Note the dummy entry at the end to give me the last-item-plus-one value, so Z ran from 38203 to 39881.

Re: passing a string to a dll

Remember that  $O(x)$  for some function  $x$  means  $K + C*x$  where  $K$  is a constant setup/teardown overhead and  $C$  is the constant-of-proportionality. When  $C$  is high, even  $n \log n$  can be slow. So the fast index gave me  $26 \log 26$ , or  $\sim 130$ , but  $C$  is very tiny. Then with an average of  $40,000/26 = \sim 1500$ , I had  $1500 * \log 1500$ , or about  $1500 * 11$  or  $\sim 16,000$  (compare this to not using the fast lookup, which would mean  $40,000 \log 40,000$  or  $\sim 640,000$ )

Although it would not apply in this case, I also kept an LRU cache of about 100 items, which gave me a major performance improvement based on the nature of the repeated queries.  
joe

On Wed, 19 Sep 2007 22:08:41 -0400, "SteveR" <srussell@xxxxxxxxxxxxxxxxxxxxxxxxxxxx> wrote:

I did not provide the number (10,000) because I wasn't asking about how else I might handle the array. I was focused on learning how to get a DLL working.

However, I actually have been wondering if I could use a second string table. I did not know that a string table could be searched by other than the stringID, though I had tried to find something. Both of these steps — creating a second table and using bsearch would be entirely new for me. Thanks for mentioning it, Joe. If you have time for a bit more detail on such an arrangement, I would definitely be interested to read it.

-----  
"Joseph M. Newcomer" <newcomer@xxxxxxxxxxxx> wrote in message [news:4f03f31va4rmlr187mkc37ilcqkauosfdl9@xxxxxxxxxxxx](mailto:news:4f03f31va4rmlr187mkc37ilcqkauosfdl9@xxxxxxxxxxxx)

Part of the reason I wasn't certain about the cost of `std::map` is that it is a balanced tree system. However, since I used to worry about all this sort of thing, you have to look at insertion cost, organization cost, and search cost. There is no one simple answer except that array is almost always wrong for something that has "thousands" (undefined? 2,000? 200,000?) in its name. Given how easy it is to use `std::map` or a hash table (I think it's `std::hash`, but I haven't used one yet, since what I've needed is sorted sets), there seems to be little reason to worry about these costs, which can be rendered very low. Even when using an array, a `CArray::SetSize(0, HUGE_ESTIMATED_NUMBER_HERE)` will do wonders for construction. Sorting is less an issue. But given the lookup happens just once, at startup time, there's a question as to where these values come from. One approach would be to make them a resource. Then you can create them, sort them, and write out a resource file that can hold them, and you can use bsearch to search

Re: passing a string to a dll

the resource.

I've done this for precanned databases and it works very well indeed.

Then the insertion

cost is 0, the sorting cost is 0 (it's all been done beforehand), and we are left only

with lookup cost, and even for reasonable thousands, bsearch works better.

The problem

with the original description is that it provides absolutely nothing to go on, just some

vague handwave about "thousands" of items. Quantities matter, certainly quantities to

within an order of magnitude should be provided; details of how the values are created

matters; details of whether or not the values need to be secured matters;

key here is that

without a lot more detail than the vague description, no solid answer can be made. But

within the vague parameters given, array sounds suspect as a strategy.

joe

On Wed, 19 Sep 2007 15:16:50 +0200, "Giovanni Dicanio"

<giovanni.dicanio@xxxxxxxxxxx>

wrote:

"Joseph M. Newcomer" <newcomer@xxxxxxxxxxxx> ha scritto nel messaggio  
[news:3qc1f39uj6cqkg3j0tr041d67saln715h0@xxxxxxxxxxx](mailto:news:3qc1f39uj6cqkg3j0tr041d67saln715h0@xxxxxxxxxxx)

The real purpose of this particular DLL is to store thousands of alphanumeric codes. Presently, I do this in a function in the exe, building a large CStringArray to which each constant is added, e.g.,  
array.Add(\_T("K34TU79456T1"))  
); When a string (code) is passed to this function, an iteration searches for that string in the array and returns a bool value accordingly. The call occurs only once during the running of

Re: passing a string to a dll

the  
exe. This has been working  
very effectively, but perhaps  
not so  
efficiently; I am not sure  
how to evaluate the  
performance other than  
that  
the results are instantaneous  
and accurate.

\*\*\*\*

This is probably a bad design. Use CMap or  
std::map, at the very least.

An array is just  
about the worst possible choice when  
thousands of keys are involved,  
since  
the time to  
search is  $O(n/2)$ , whereas for hash tables it  
can be  $O(k)$  for some small  
integer  $k$ ;  
std::map is fairly fast, although I'm not sure  
what its complexity is (I  
haven't checked),  
but it going to be faster than  $O(n/2)$ .

My understanding of the OP's problem is the following:

- o He stores strings into an array; these strings represent valid codes.
- o This array is initialized only once in program life-cycle, at startup.
- o He has a function like so:

```
bool IsValidCode( string codeToCheck )
```

that searches the input string in the valid-codes array; if the  
string  
is found, then it represents a valid code, and the function  
returns true;  
else the function returns false.

If my understand is correct, I don't very much agree with  
what Joe  
suggests  
(maybe I have not understood Joe well).

Re: passing a string to a dll

In fact, I think that using the array is just fine in that context. The array is initialized only once to store the strings representing valid codes.

The trick here IMHO is to be sure that the array is *\*sorted\**. In fact, if the array is sorted, then a simple *\*binary search\** with a  $O(\log(n))$  asymptotic complexity would be just fine. And I think it would be even better than a map, which I think has worse performance than binary-search's  $O(\log(n))$ .

Moreover – IIRC, but take it with a grain of salt – I think that `std::map` is actually *\*not\** a hash-map, but a BST (binary search *\*tree\**). I think that the real hash-map is `std::hash_map`.

But, again, I think that sorted array + binary search would be just fine...

You would need to say more about this list of codes. Are they some attempt to do license keys? If so, it would take most 14-year-olds under ten minutes to crack it.

I completely agree with Joe on this point.

If I move these strings into the DLL function, I will be happy to get them out of the exe; but what impact does this have on performance, memory consumption or anything else? Am I gaining or losing anything?

\*\*\*\*\*

Zero. Putting them in a DLL has effectly zero impact. Well, there's a

Re: passing a string to a dll

small cost to load

Maybe the effect would be to make it easier for the cracker to break the protection: the DLL substitution with a cracked DLL that just returns OK when codes are checked would be just fine :(

Otherwise, you would use LPCTSTR in the usual (old-fashioned, error-prone, possibly-storage-leaking) way it has always been used.

There are contexts where LPCTSTR (or better const wchar\_t\* or even BSTR IMHO, as COM does) are OK. For example: if the DLL has a C \*interface\* (it's OK to use a string class \*inside\* the DLL, of course).

BTW: BSTR is even better than wchar\_t\*, because with wchar\_t\* it requires O(n) to find the string length, while with BSTR is just optimal O(1), being the BSTR length-prefixed.

Giovanni

Joseph M. Newcomer [MVP]  
email: newcomer@xxxxxxxxxxxxx  
Web: <http://www.flounder.com>  
MVP Tips: [http://www.flounder.com/mvp\\_tips.htm](http://www.flounder.com/mvp_tips.htm)

Joseph M. Newcomer [MVP]  
email: newcomer@xxxxxxxxxxxxx  
Web: <http://www.flounder.com>  
MVP Tips: [http://www.flounder.com/mvp\\_tips.htm](http://www.flounder.com/mvp_tips.htm)