Unsupervised: Prelert's Machine Learning Blog

Filter by Interest

IT/Security IT Ops/APM Math & Code Nerds

A look at std::string implementations in C++

Posted by David on 9/10/15 11:44 AM

I've previously compared the implementations of the container classes in the C++ standard library. Following the recent release of g++ version 5.1 I thought it would be interesting to do the same for the std::string class.

Search

Subscribe to Updates

Email*

Popular Posts

- STL Container Memory Usage

the C++98 and C++11 standards. The C++98 standard mandates:

1. The character type must be a plain old data (POD) type
2. size() <= capacity()
3. size() <= max_size()
4. std::string must have random access iterators
5. References, pointers and iterators must only be invalidated by the following:
    a. When the std::string is passed as a non-const reference to a standard library function
    b. When a non-const std::string member other than operator[](), at(), begin(), rbegin(), end() and rend() is called, ***except*** the first call to non-const operator[](), at(), begin(), rbegin(), end() or rend()
6. reserve(n) with n < capacity() is a non-binding shrink request
7. std::string must have a constant time, non-throwing swap()
8. The const version of operator[](size()) is valid and returns charT(); it is undefined what the non-const version returns for an index of size()
9. c_str() returns a zero-terminated array

1. The character type must be a POD type
2. size() <= capacity()
3. size() <= max_size()
4. std::string must have random access iterators
5. References, pointers and iterators may only invalidated by the following:
    a. When the std::string is passed as a non-const reference to a standard library function
    b. When a non-const std::string member other than operator[](), at(), front(), back(), begin(), rbegin(), end() and rend() is called
6. reserve(n) with n < capacity() is a non-binding shrink request
7. std::string must have a constant time, non-throwing swap()
8. Both const and non-const versions of operator[](size()) are valid calls and return charT(); for the non-const version the caller must not modify the charT() when the index is size()
9. c_str() returns a zero-terminated array
10. data() returns a zero-terminated array
11. The underlying characters are stored contiguously

Important: the lists above are ***loose summaries*** of what the C++ standard says, focusing on the aspects most relevant to this post. If you need the full list please read section 21 of the C++ standard.

The extra constraints imposed by C++11 were driven by this proposal from 2008, which aimed to improve opportunities for efficient concurrency. The extra constraints effectively rule out copy-on-write implementations of std::string.

Copy-on-write was a popular technique in the

same block of memory to store it until such time as one of the strings was modified. The C++98 standard even goes as far as to say with regard to strings, "These rules are formulated to allow, but not require, a reference counted implementation."

Unfortunately it turned out that copy-on-write had serious drawbacks in multi-threaded programs. Even strings used within a single thread have to perform expensive serialisation when testing the reference count, and they have to test the reference count even for seemingly innocuous operations such as the non-const operator[](). For a detailed demolition of the claim that copy-on-write is the fastest way to implement strings in multi-threaded environments, see item 16, "Lazy Optimization, Part 4: Multithreaded Environments" and Appendix B in Herb Sutter's 2001 book "More Exceptional C++".

The most popular technique for implementing std::string nowadays is the "small string optimisation" (SSO). With this technique, each string object contains a small buffer so that strings only a few characters in length can be stored entirely within the string object and do not require a separate heap memory allocation. Management of heap memory is complex and much slower than allocation of stack memory, so for short strings used within a single block scope

One key decision that the C++ standard still leaves up to std::string implementers is the growth strategy – how much should the string capacity grow when characters are appended? The most naïve way would be to increase the capacity only to accommodate the character(s) being appended. However, this would be diabolically inefficient in the case of building up a large string one character at a time, because each new character would cause a memory reallocation and the need to copy all the existing characters to the new buffer.

In "More Exceptional C++", the sub-section entitled "Aside: What's the Best Buffer Growth Strategy?" within item 13, "Lazy Optimization, Part 1: A Plain Old String", concludes that an exponential growth strategy that increases string capacity by 1.5 times at each reallocation is best. This is based on a 1998 paper "Why Are Vectors Efficient?" by Andrew Koenig, but std::string is very similar to std::vector in this respect. Andrew Koenig's paper isn't freely available on the internet, but this UseNet post talks about the finding. The biggest growth factor that allows freed space to be reused by subsequent allocations is (1 + sqrt(5))/2: the golden ratio. I'm not convinced the argument holds today until you reach enormous string sizes, because modern allocators are considerably more complex than those that existed in 1998 and generally allocate memory

of the allocator memory pools. Allocations made by other threads in a multi-threaded program could also play havoc with the reasoning.

Another decision that the C++ standard leaves to implementers is whether the reserve() method should reduce capacity. For example, if I have a string with size 10 and capacity 100 and call reserve(20) on it, what happens to the capacity? Requests to reduce string capacity are non-binding, so it's worth knowing what the popular implementations actually do.

Anyway, enough theory: let's look at the details of some real-world implementations. The six std::string implementations I've investigated are:

1. Visual Studio 2010
2. Visual Studio 2013
3. g++ 3.4-4.9
4. g++ 5.1+
5. libc++ from LLVM
6. Apache

I'm only going to consider the 64 bit versions, although if you're still building 32 bit code you can probably work out what the equivalents will be: all the size_t's and pointers will halve in size.

Starting with the Visual Studio 2010 and 2013 strings, they both use the small string optimisation. Strings up to 15 characters in length are stored within the string object itself. Each string object has members for size and capacity, plus a union of size 16 bytes that stores

Well, in Visual Studio 2013 that's correct, but Visual Studio 2010 isn't quite so efficient.

The underlying std::basic_string template lets you specify which allocator it should use. If you don't specify one, a default template argument means you're implicitly choosing std::allocator<charT> where charT is the type of character the string holds. This will generally boil down to using malloc() and free(). Visual Studio 2010 strings use a data member to store the allocator, and in the case of std::allocator<T> this adds 8 bytes to the overall object size, making Visual Studio 2010's strings 40 bytes in size. However, an instance of std::allocator<T> has no non-static data members and in Visual Studio 2013 use of template specialisation on (undocumented) base classes of the string enables this allocator data member to be dropped from them altogether by taking advantage of the empty base optimisation. If you've read my other post about STL container memory usage you'll recognise this as exactly the same optimisation Microsoft made to the container classes between Visual Studio 2010 and Visual Studio 2013.

Visual Studio 2010 and 2013 both use a growth factor of 1.5 when increasing capacity (Herb Sutter works for Microsoft so maybe it's not surprising they've followed his recommendation), and both their reserve() methods can reduce the

Next, let's move on to the strings in GNU's libstdc++. This library has been through several iterations over the years, but the current libstdc++ version 3, usually deployed as libstdc++.so.6, has been used since g++ version 3.4, which was first released in 2004. I won't consider anything older than this.

The default std::string implementation in libstdc++ prior to g++ 5.1 used a copy-on-write strategy. This meant that for several years after the release of the C++11 standard g++'s strings did not comply with the standard: the designers of libstdc++ considered backwards compatibility more important than strict standards conformance (and having experienced the switch from libstdc++.so.5 to libstdc++.so.6 I have to say I agree with their decision). Starting with g++ 5.1 there is a second std::string implementation available in libstdc++.so.6 in parallel to the old one. At compile time this is enabled by defining the macro _GLIBCXX_USE_CXX11_ABI. The really clever thing the libstdc++ developers have done is to enable the same shared library to supply both the old and new implementations, such that it will work with programs compiled either with or without the _GLIBCXX_USE_CXX11_ABI macro defined. From now on I'll refer to the new string implementation enabled by _GLIBCXX_USE_CXX11_ABI as the g++ 5.1+ string, even though this isn't strictly correct

The copy-on-write g++ string is typical of copy-on-write implementations, in that sizeof(std::string) is just 8 bytes: its only data member is a struct that inherits from the allocator and contains a pointer to a (potentially) shared representation. Because the allocator is stored as a base class it can take advantage of the empty base optimisation, so allocators with no data members don't increase the object size. However, it's worth noting that if you do instantiate a std::basic_string template with an allocator that needs data then this overhead gets added to every std::string object rather than the shared representations. This makes sense as you need to know the allocator before you can allocate the shared representation. The shared representation then contains a size, capacity and a reference count in addition to the space for the string characters and a zero terminator.

The reference count for the shared representation is made thread-safe through the use of atomic operations rather than mutexes. This benefits performance – in the measurements I took on Linux for this post mutexes were about 5 times slower than atomic operations for guarding simple increments/decrements of integers.

One design decision that a copy-on-write implementer has to make is what happens when a string reverts to being empty. Copy-on-write implementations tend to have a global object

allocation of an empty representation – they simply point to the global one. But should a string that was not empty revert back to the global empty string representation when it becomes empty? In the case where the representation is not shared, the obvious answer is no: you want to be able to retain the non-zero capacity you've already got in case you append to the string later. In the case where the representation is shared the best thing would be to revert to the global empty representation. Unfortunately the g++ copy-on-write string doesn't do this leading to perverse performance issues when clearing strings that have been copied.

The growth factor for the g++ copy-on-write string is 2. If you append characters one at a time you'll see the capacity follow the sequence 1, 2, 4, 8, 16, 32, etc. There is some cleverness built in though, because once you get close to the assumed page size of the operating system, 4096 bytes, the capacities are set such that the representation plus the malloc() overhead will fit into an exact number of pages.

The reserve() method does reduce capacity as much as possible, and the shrink_to_fit() method is supported providing you're using at least g++ 4.5 and are using the compiler option -std=gnu++0x or some other -std value that implies C++11 or higher.

strings up to 15 characters long avoid heap allocations. The object layout is slightly different, with the capacity being stored in the small string buffer union rather than the data pointer. This can be done because capacity is known for small strings. It means the data pointer is valid for both short and long strings, which in turn means the c_str() and data() methods don't have to check whether the string is using heap memory before returning their answer. On the flip side, the capacity() method has to contain more logic than Visual Studio's. Empty allocators don't add anything to the total string size through use of the empty base optimisation – the data pointer is in a struct that inherits from the allocator – making sizeof(std::string) 32.

The growth factor for the g++ 5.1+ string is 2, with no cleverness (or premature optimisation depending on your viewpoint) regarding fitting to operating system page sizes. The reserve() method reduces capacity as much as possible, and the shrink_to_fit() method is supported.

One gotcha I should mention about the _GLIBCXX_USE_CXX11_ABI macro is that some distributions of g++ 5.1 have it set to 0 by default, for example Fedora 22. When building C++ programs with the packaged compiler on Fedora 22 you need to add the compiler flag -D_GLIBCXX_USE_CXX11_ABI=1 if you want to take advantage of the new string

we'll move on to look at a string implementation that's clearly been designed from the ground up with space efficiency in mind. The std::string from LLVM's libc++ is just 24 bytes in size, yet allows strings up to 22 characters long to be stored in the object using the small string optimisation. How do they do that?

The answer is that two completely different layouts for the 24 byte string object are defined as follows:

```
struct __long
{
    size_type __cap_;
    size_type __size_;
    pointer   __data_;
};
```

and:

```
struct __short
{
    union
    {
        unsigned char __size_;
        value_type __lx;
    };
    value_type __data_[__min_cap];
};
```

That looks easy enough. For a long string you need size, capacity and a pointer to the memory storing the data. For a short string you need just size and data, as the capacity is known, and you know the size will fit into an 8 bit number because by definition the string is short. All that remains is to know whether a string is long or short without having to add any more data

__size_ member does the size of the string, but double the true size. As a result, the lowest bit of the __size_ member of a __short struct will always be 0. In the case of __long, the __cap member doesn't store the true capacity, but true capacity bitwise-or'd with 1. Capacity will always be adjusted to be one less than a multiple of 16, but then allocations are done for 1 more than the capacity, so using the lowest bit as a flag doesn't lose information. The long and short of it (geddit) is that the lowest bit of the very first byte in the string object is 1 when the long representation is used and 0 when the short representation is used. (I've oversimplified that a little, as it's different on big endian architectures and there is an alternative memory layout that can be enabled using macros and means that the pointer returned by c_str() and data() is always aligned on a word boundary. However, hopefully you'll get the gist from what I said.)

Allocators with no data members add no space overhead through use of the compressed pair technique: std::string is a compressed pair whose first member is a union of __long, __short and an array of size_t that means sizeof(std::string) is always a multiple of sizeof(size_t) and whose second member is the allocator. Boost also contains a compressed pair class if you want to read more about this technique.

The libc++ std::string will reduce capacity if appropriate following a call to reserve() and

the Apache C++ Standard Library. This has been moved to the attic now, but since I have the figures I might as well share them, if only to demonstrate that C++ standard libraries have advanced over the last 10-15 years.

Like the older g++ implementation, the Apache std::string uses copy-on-write with reference counts maintained using atomic operations. Also like the g++ copy-on-write implementation the size of the std::string object itself is just 8 bytes, as it just has to point to the shared representation.

Despite this, the Apache std::string is quite inefficient in its use of memory, especially for smaller strings. Default constructed strings point to a global empty representation with capacity 0, so are very fast to create. When the string grows beyond 0 size, the capacity increase is governed by both a multiplier and an absolute minimum. The absolute minimum is supposed to be specified by the _RWSTD_MINIMUM_STRING_CAPACITY macro, which defaults to 128. In my opinion this is too large for a minimum string capacity. Programs that create many small strings end up wasting a lot of memory. There's a twist in the tale however. Due to the age of the Apache library there's actually a bug in the way this minimum capacity is applied, and this means that modern compilers pick up the macro _RWSTD_MINIMUM_NEW_CAPACITY instead, which

make consideration of candidate template specialisations conform to the standard was made in g++ 3.4 (in 2004) and Solaris Studio 12.4 (in 2014).

Once the string capacity has increased enough for the growth factor to take precedence over the absolute increment, the growth factor is 1.618: very close to the golden ratio. From Andrew Koenig's paper that I mentioned earlier, this is the maximum growth factor that allows larger memory allocations to fit into the space left behind by the sum of the smaller ones freed up as the string grew. As I said before, it's a nice theory but I think it will only hold in the limit with a modern memory allocator.

The Apache copy-on-write string doesn't suffer from the g++ issue with clearing shared strings – these revert to the global empty representation, so such clears aren't unexpectedly expensive.

Finally, the Apache std::string never reduces capacity in response to a reserve() call and does not provide a shrink_to_fit() method either – if you want to reduce capacity you have to use a trick like:

```
std::string(myString, 0, myString.length()
```

If you've read item 17 in an early printing of Scott Meyers's 2001 book "Effective STL" and thought:

```
std::string(myString).swap(myString);
```

this was fixed in later printings according to the errata list – maybe I should wait until longer after publication before buying C++ books! (Search for "the swap trick" in the errata list to read what Scott Meyers has to say about it.)

For reference, here's a table summarising everything:

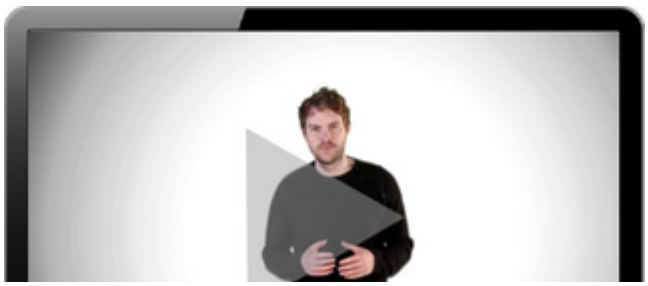
	Visual Studio 2010	Visual Studio 2013	GNU libstdc++ g++ 3.4-4.9	GNU libstdc++ g++ 5.1+	LLVM libc++	Apache stdcxx
sizeof(std::string)	40	32	8	32	24	8
Heap structure size	capacity() + 1	capacity() + 1	capacity() + 25	capacity() + 1	capacity() + 1	capacity(+ 26
Uses small string optimisation?	Y	Y	N	Y	Y	N
Uses copy-on-write?	N	N	Y	N	N	Y
Initial empty representation shared?	N	N	Y	N	N	Y

representation?						
Largest capacity() before heap allocation	15	15	0 †	15	22	0
Smallest non-zero capacity()	15	15	1	15	22	32 *or* 128 ‡
capacity() growth strategy	* 1.5 Minimum 16	* 1.5 Minimum 16	* 2 Above 4096 bytes round up to page size less assumed malloc() overhead	* 2 No special rules	* 1.5 Minimum 16	* 1.618 Minimum 32 *or* 128 ‡
reserve() can reduce capacity()	Y	Y	Y	Y	Y	N
Has shrink_to_fit?	Y	Y	C++11 mode only from g++ 4.5	C++11 mode only	Y	N

+ Depending on whether the compiler considers template specialisations defined below the point of use in a translation unit (which they shouldn't)

(If you're using this page as a reference you can link directly to the summary table at http://info.prelert.com/blog/cpp-stdstring-implementations#summarytable.)

In conclusion, the different string implementations tell a story of the compilers and platforms they're most associated with. The GNU strings have evolved slowly over the years being careful to maintain backwards compatibility rather than blindly following the C++ standard. The Visual Studio strings can be re-implemented for every release, so have been able to move to best practice (the small string optimisation) faster. But interestingly, only recently has the default allocator been optimised out of the size. The libc++ std::string struts its modernity with every scrap of efficiency eked out. And the Apache implementation shows its age and lack of attention fully justifying the decision of the libc++ developers not to base their work on it.



Automated Anomaly Detection: Under the Hood

To learn about the algorithms and statistical modeling in our anomaly detection software, check out this short (4 min) technical video.

View More Developer Blogs Like This

Topics: Developer

Paul Hampson 30/9/2015 00:02:05

Maybe I'm being nitpicky, but how does "std::string must have a constant time, non-throwing swap()" work with small-string optimisation, where swap must clearly swap each

Is this "too small to worry about", or am I overlooking some constant-time swap method for small-string optimised basic_string?

Reply to *Paul Hampson*

David Roberts 30/9/2015 05:46:29

Yes, I think you're right that this is "too small to worry about" Paul.

When a string that's long is swapped in an implementation that isn't reference counted, it has to swap three member variables: capacity, size and pointer to data. For a 64 bit implementation these will all be 64 bit values, so 24 bytes needs to be swapped in total.

For the libc++ implementation of the short string optimisation, swapping a short string is identical - it just has to swap three 64 bit values. As far as swap() is concerned it doesn't matter that the interpretation of these three values is different for long and short strings. So, for libc++, swap() is identically efficient for long and short strings.

The g++ and Visual Studio implementations aren't quite so efficient, as they don't pack the content quite as much. But even then, for the case of swapping short strings they just

operation. (We think this is guaranteed by the fact that the character type of the string must be a POD.)

Hope this helps,

David

Reply to *David Roberts*

First Name

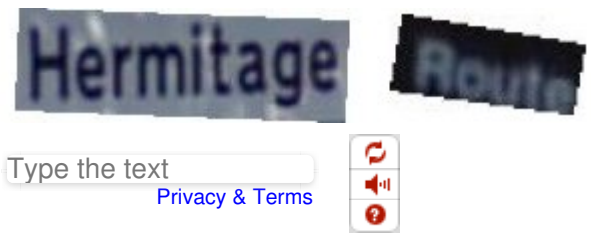
Last Name

Email*

Website

Comment*

Subscribe to follow-up comments for this post



Type the text

Privacy & Terms

Submit Comment