An informal comparison of the three major implementations of std::string

devblogs.microsoft.com/oldnewthing/20240510-00

May 10, 2024

Raymond Chen

[Note: This article has been updated since original publication.]

We saw some time ago that the three major implementations of `std::string` are all quite different. To summarize:

```
// gcc
struct string
{
    char* ptr;
    size_t size;
    union {
        size_t capacity;
        char buf[16];
    };

    bool is_large() { return ptr != buf; }
    auto data() { return ptr; }
    auto size() { return size; }
    auto capacity() { return is_large() ? capacity : 15; }
};

// msvc
struct string
{
    union {
        char* ptr;
        char buf[16];
    };
    size_t size;
    size_t capacity;

    bool is_large() { return capacity > 15; }
    auto data() { return is_large() ? ptr : buf; }
    auto size() { return size; }
    auto capacity() { return capacity; }
};

// clang
union string
{
    struct {
        size_t capacity;
        size_t size;
        char* ptr;
    } large;

    struct {
        unsigned char is_large:1;
        unsigned char size:7;
        char buf[sizeof(large) - 1];
    } small;

    bool is_large() { return small.is_large; }
    auto data() { return is_large() ? large.ptr : small.buf; }
    auto size() { return is_large() ? large.size : small.size; }
```

```
    auto capacity() { return is_large() ? large.capacity : sizeof(large) - 2; }
};
```

Note that these implementations above are for expository purposes only. The actual implementations are far more complicated. (For example, the real implementations are uglified, and they have to store the allocator somewhere.)

Update: In the original version of this article, I got the sense of the “small/large” bit backward in the clang implementation. This in turn led to redoing the code generation and new code golfing results.

We’ll compare these versions based on the complexity of some commonly-used operations.

Detecting whether the string is small or large is a single member comparison with msvc and clang, but on gcc, it involves comparing a member against the address of another member, so it will take an extra instruction to calculate that address.

gcc is_large	msvc is_large	clang is_large

`lea rdx, [rcx].buf`
`cmp rdx, [rcx].ptr`
`jnz large`
	`cmp [rcx].capacity, 15`
	`ja large`
		`test [rcx].is_large, 1`
		`jnz large`

Note: gcc could have shaved an instruction by reordering the members so that the `buf` comes first (thereby avoiding the need to calculate its address). On the other hand, it increases the cost of accessing `ptr` on some processors: On the x86 family, it forces a larger encoding because the offset is nonzero. On the Itanium, it requires two instructions because the Itanium cannot perform an offset load in a single instruction. On most other processors, the offset can be folded into the load instruction at no extra cost. My guess is that gcc biased their design to optimize for x86.

On the other hand, gcc wins the race to access the `data()`, since the `ptr` is always valid, and that’s probably why they chose their design.

gcc data()	msvc data()	clang data()[1]

`mov rdx, [rcx].ptr`	`lea rdx, [rcx].buf`
	`cmp [rcx].capacity, 15`
	`cmova rdx, [rdx]`
		`lea rdx, [rcx].small.buf`
		`test [rcx].small.is_large, 1`
		`jz @1`
		`mov rdx, [rcx].large.ptr`
		`@1:`

The clang implementation also has extra work to calculate the size.

gcc size()	msvc size()	clang size()[2]

`mov rdx, [rcx].size`	`mov rdx, [rcx].size`	`movzx eax, [rcx].small.is_large`
		`test al, 1`
		`jz @1`
		`mov rax, [rcx].large.size`
		`jmp @2`
		`@1: shr eax, 1`
		`@2:`

A special case of checking the size is checking whether the string is empty.

gcc empty()	msvc empty()	clang empty()[3]

`cmp [rcx].size, 0`
`jz empty`
	`cmp [rcx].size, 0`
	`jz empty`
		`movzx eax, [rcx].small.is_large`
		`test al, 1`
		`jz @1`
		`mov rax, [rcx].large.size`
		`jmp @2`
		`@1: shr eax, 1`
		`@2: test rax, rax`
		`jz empty`

The capacity comes into play behind the scenes when extending the string. For example, `append(char)` can do a fast-append if there is excess capacity, and delegate to a function call if the capacity needs to be increased. Here, msvc has an edge.

gcc capacity()	msvc capacity()	clang capacity()

`lea rax, [rcx].buf`
`cmp rax, [rcx].ptr`
`je @1`
`mov rax,`
`[rcx].large.capacity`
`jmp @2`
`@1: mov eax, 15`
`@2:`
	`mov rax,`
	`[rcx].capacity`
		`test [rcx].small.is_large, 1`
		`mov eax, 22`
		`jz @1`
		`mov rax,`
		`[rcx].large.capacity`
		`@1:`

The clang implementation does have an edge in terms of memory usage: Despite an overall smaller size, it has a larger small-string capacity in 64-bit mode.

sizeof / SSO capacity	gcc	msvc	clang
32-bit mode	24 / 15	24 / 15	12 / 11
64-bit mode	32 / 15	32 / 15	24 / 22

If you `reserve()` a lot of space for a string, but use only a little bit of it, and then call `shrink_to_fit()`, you can potentially get into a mixed state where the string is allocated externally (as if it were a large string), even though the capacity is smaller than the capacity of a small string.

The msvc implementation uses the capacity to detect whether it is using the small string optimization, so this mixed state is illegal for msvc, and it must convert large strings to small strings if `shrink_to_fit()` shrinks the string below the small-string threshold.

The gcc and clang implementations allow external allocations to have a small capacity. Nevertheless, both gcc and clang force the conversion of externally-allocated strings to small strings if they shrink below the small-string threshold.

Update: A previous version of this article erroneously said that `shrink_to_fit()` is a nop on gcc.

One final point of comparison is how the three implementations deal with static initialization.

gcc	msvc	clang
`{ buf, 0, { 0 } }`	`{ { 0 }, 0, 15 }`	`{ 0, 0, 0, ... }`

A statically-initialized empty string in gcc consists of a pointer to the internal buffer, a constant 0 (size), and a bunch of zeroes (buf). The presence of a pointer introduces a relocation into the data segment and silently messes up string's `constexpr`-ness.

Statically-initialized empty strings in msvc and clang consist of integer constant data; no pointers. This means no relocations and a better shot at `constexpr`-ness.

Okay, so let's summarize all this information into a table.

	gcc	msvc	clang
is_large	slower	faster	faster
data()	fast	slower	slower
size()	fast	fast	much slower
empty()	fast	fast	much slower[3]
capacity()	slowest	fast	slower
32-bit size	24	24	12
64-bit size	32	32	24
32-bit SSO capacity	15	15	11
64-bit SSO capacity	15	15	22
ABI supports mixed state?	yes	no	yes
implementation uses mixed state	no	forbidden	no
Static initialization	relocation	no relocation	no relocation

[1] I don't see clang generating this slightly smaller alternative

```
lea rdx, [rcx].small.buf
test [rcx].small.is_large, 1
cmovnz rdx, [rcx].large.ptr
```

perhaps because the `cmov` instruction always reads from its second parameter even if the value is not used, and there might be a store-to-load forwarding penalty because in the case of a small string, the read is unlikely to match the size of the previous write.

[2] I don't see clang generating this slightly smaller alternative

```
movzx eax, [rcx].small.is_large
shr eax, 1
jnc @1
mov rax, [rcx].large.size
@1:
```

probably because "shift right and look at carry" is not something natively expressible in C++. If you really went for it, you could also fold in a `cmov`.

```
movzx eax, [rcx].small.is_large
shr eax, 1
cmovc rax, [rcx].large.size
```

[3] My hand-golfed version of clang `empty()` brings the performance of clang `empty()` to be comparable to gcc and msvc:

```
cmp byte ptr [rcx].small.is_large, 0
jz empty
```

The trick here is that since clang always uses SSO when possible (no mixed state), the `is_large` is sufficient to tell us whether the string is empty. A large string is never empty, and it will have the `is_large` bit set, so a large string will never have zero as its initial byte. A small string has the `is_large` bit clear and the string size in the remaining bits, so comparing the entire byte against zero tests the size and the `is_large` bit simultaneously.

While it's true that clang always uses SSO when possible, it's still a valid state for a large string to be empty, because it might have a large capacity but no contents. So we just have to get the size and test it against zero. (Though we can cheat and omit the shift-right since zero shifted left or right by one is still zero.)

```
movzx eax, [rcx].small.is_large
test al, 1
cmovnz rax, [rcx].large.size
test rax, rax
jz empty
```