Inside STL: The string

devblogs.microsoft.com/oldnewthing/20230803-00

August 3, 2023

Raymond Chen

You might think that a `std::string` (and all of its friends in the `std::basic_string` family) are basically a vector of characters internally. But strings are organized differently due to specific optimizations permitted for strings but not for vectors.

The starting point for a `std::basic_string` is this:[1]

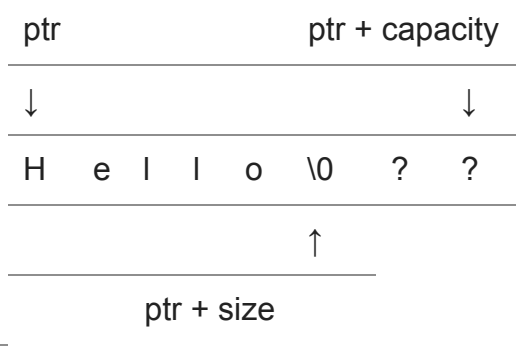
```
template<typename T>
struct basic_string
{
    T* ptr;
    size_t size;
    size_t capacity;
};
```

The `ptr` is a pointer to the beginning of the string contents.

The `size` is the number of characters in the string, not including the null terminator.

The `capacity` is the string capacity, not including the null terminator.

The picture for this simplified version is as follows:



The `ptr` points to the start of the allocation. The first `size` characters of the allocation contain the string contents. The next character contains a null terminator. And then there are `capacity - size` additional uninitialized characters.

Since the `capacity` does not include the null terminator, the total size of the allocation is `capacity + 1` characters. In the diagram, it looks like the allocation "sticks out" beyond `ptr + capacity`. The extra character is for the potential null terminator needed if string grows so that the size equals the capacity.

However, there is an added wrinkle: The *small string optimization*.

Recall that vectors were required to keep the memory allocation entirely external to the vector object itself due to the requirement that moving a vector does not invalidate iterators or references.[2] However, the standard does not impose the same requirement on strings.

Let's take advantage of this by allocating a small buffer *inside the* `std::basic_string` for the (hopefully) common case that the string is relatively short. Doing so avoids any allocation at all.

Here's a second version:

```
template<typename T>
struct basic_string
{
    static constexpr size_t BUFSIZE = 8;
    T* ptr;
    size_t size;
    size_t capacity;
    T buf[BUFSIZE]; // special storage for short strings
};
```

If the string capacity is at most some fixed value (here, 8 −1 = 7), then we can point `ptr` to the internal `buf` instead of a separate heap allocation.

For this small string "Hello", the `ptr` points not to a separate heap allocation but to the `buf` member. And you can detect whether the small string optimization is in effect by seeing whether `ptr == buf`. If the `ptr` points to the internal `buf`, then you are using the small string optimization. Otherwise, it's a pointer to an external allocation.

In particular, the small string optimization means that an empty string can be created without any heap allocations.

It is here that the three major implementations of the C++ standard library go their different ways. Each implementation has to balance a variety of things:

- A bigger internal buffer lets you avoid more external allocations, but it also consumes more memory, which is wasted in the case where the string doesn't fit in the internal buffer.
- The tricks generally reduce memory usage per string but increase code complexity. If there are a lot of inlined string operations, the extra code generation may exceed the data memory savings.

The gcc standard library realizes that if you are using the internal buffer `buf`, then you don't need to record its `capacity` explicitly, because you already know its capacity: It's `BUFSIZE - 1`. Therefore, gcc's implementation overlays the the `capacity` with the `buf` since only one of them is needed at a time. The size of the internal buffer is chosen so that it holds as many characters as fit into 16 bytes (minus 1, for the null terminator). This means you can hold up to 15 `char`s or up to 7 UTF-16 code units.

```
template<typename T>
struct basic_string
{
    static constexpr size_t BUFSIZE = 16 / sizeof(T);
    T* ptr;
    size_t size;
    union {
        size_t capacity;
        T buf[BUFSIZE]; // special storage for short strings
    } storage;
};
```

By comparison, the Microsoft standard library doesn't use the `ptr` to detect whether the internal buffer is in use. Instead, it looks at the `capacity`: If the capacity is `BUFSIZE - 1`, then you're using the internal buffer. If the capacity is greater than or equal to `BUFSIZE`, then you're using an external buffer.[3] The Microsoft standard library also chooses an internal buffer of 16 bytes. The result is something like this:

```
template<typename T>
struct basic_string
{
    static constexpr size_t BUFSIZE = 16 / sizeof(T);
    union {
        T* ptr;
        T buf[BUFSIZE];
    } storage;
    size_t size;
    size_t capacity;
};
```

The third major implementation is clang, and it takes the optimization to a much further level. There are basically two versions of the structure, distinguished by the lowest-order bit of the first byte (on little-endian systems).

If the lowest-order bit of the first byte is clear, then the `basic_string` is using an external allocation. If the bit is set, then it is using the small-string optimization. Let's look at the large case first.

```
template<typename T>
struct basic_string_large
{
    size_t capacity; // always multiple of 2
    size_t size;
    T* ptr;
};
```

The implementation rounds up the requested capacity so that it is always an even number and therefore has its bottom bit clear, indicating that this is a string with an external allocation.

Next is the small string layout:

```
template<typename T>
struct basic_string_small
{
    unsigned char is_small:1; // always set for small strings
    unsigned char size:7;
    T buf[BUFSIZE];
};
```

For a small string, the first byte has its bottom bit set, and the remaining bits are the size. (Another way of looking at this is that the first byte contains the size doubled plus 1.) The `BUFSIZE` is chosen so that the `basic_string_small` is no larger than a `basic_string_large`. For a 64-bit system, it means that `BUFSIZE` is 23 if `sizeof(CharT) == 1` and 11 if `sizeof(CharT) == 2`. In all cases, `BUFSIZE` is less than 128, so we can always squeeze the size of a short string into 7 bits.

This sneaky trick lets clang shrink a `std::string` down to 24 bytes on 64-bit systems, while still allowing 8-bit strings up to 22 characters and 16-bit strings up to 10 characters to be treated as short.

The Visual Studio debugger contains a visualizer to view the contents of `std::string` and `std::wstring` more conveniently, but if you need to dig out the contents manually, here's how you can do it with the Microsoft implementation of the standard library. Some of the extra layers of wrapping are due to the usual optimization of sneaking the allocator (which is almost always an empty class) into the base class of a compressed pair.

```
0:000> ?? s
class std::basic_string<char,std::char_traits<char>,std::allocator<char> >
   +0x000 _Mypair          :
std::_Compressed_pair<std::allocator<char>,std::_String_val<std::_Simple_types<char>
>,1>
0:000> ?? s._Mypair
class
std::_Compressed_pair<std::allocator<char>,std::_String_val<std::_Simple_types<char>
>,1>
   +0x000 _Myval2          : std::_String_val<std::_Simple_types<char> >
0:000> ?? s._Mypair._Myval2
class std::_String_val<std::_Simple_types<char> >
   +0x000 _Bx              : std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x010 _Mysize          : 5
   +0x018 _Myres           : 0xf
0:000> ?? s._Mypair._Myval2._Bx
union std::_String_val<std::_Simple_types<char> >::_Bxty
   +0x000 _Buf             : [16]  "Hello"
   +0x000 _Ptr             : 0x0000006f`6c6c6548  "--- memory read error at address
0x0000006f`6c6c6548 ---"
   +0x000 _Alias           : [16]  ""
```

Our name	MSVC name	Notes
`storage.ptr`	`_Bx._Ptr`	Bx = buffer
`storage.buf`	`_Bx._Buf`	
`size`	`_Mysize`	
`capacity`	`_Myres`	res = reserve

There's also an `_Alias` that is another name for `_Buf` which MSVC keeps around for debugger backward compatibility but which is not used by the implementation.

In the above case, we are looking at a `std::string`, so the internal buffer `_Buf` holds 16 characters. We see that the capacity is `0xf`, so the string contents are in the internal buffer, and we see it right there in the `_Bx._Buf`.

```
0:000> ?? t
class std::basic_string<wchar_t,std::char_traits<wchar_t>,std::allocator<wchar_t> >
   +0x000 _Mypair          :
std::_Compressed_pair<std::allocator<wchar_t>,std::_String_val<std::_Simple_types<wcha
 >,1>
0:000> ?? t._Mypair
class
std::_Compressed_pair<std::allocator<wchar_t>,std::_String_val<std::_Simple_types<wcha
 >,1>
   +0x000 _Myval2          : std::_String_val<std::_Simple_types<wchar_t> >
0:000> ?? t._Mypair._Myval2
class std::_String_val<std::_Simple_types<wchar_t> >
   +0x000 _Bx              : std::_String_val<std::_Simple_types<wchar_t> >::_Bxty
   +0x010 _Mysize          : 0xb
   +0x018 _Myres           : 0xf
0:000> ?? t._Mypair._Myval2._Bx
union std::_String_val<std::_Simple_types<wchar_t> >::_Bxty
   +0x000 _Buf             : [8]  "????"
   +0x000 _Ptr             : 0x00000277`e45f9320  "Hello there"
   +0x000 _Alias           : [8]  " ???"
```

In this case, we dump a `std::wstring`. The `_Myres` is `0xf` which is greater than 7, so the `_Bx._Ptr` points to the data.

When I'm dumping these structures, I don't look at the `_Myres`. I just dump the `_Bx`, and either the `_Buf` or `_Ptr` will hold the data, and the other will be garbage. So I ignore the garbage.

I may not have mentioned it clearly at the start, but the primary purpose of this series is to provide enough information that you can fish out the contents of these standard library data structures when you find them in a crash dump.

Bonus chatter: Older versions of gcc's standard library used reference-counted strings with copy-on-write. This changed when C++11 required additional operations to preserve iterators such as `c_str()`, `data()`, and `operator[]`. Under the old rules, this code exhibited undefined behavior:

```
void copy_first_char_to_second(std::string& dest, const std::string& source)
{
    dest[1] = source[0];
}

void example()
{
    std::string oops("xy");
    copy_first_char_to_second(oops, oops);
}
```

Read the linked paper for details.

[1] Throughout, I present a description of the data structures that is logically equivalent to what the three major libraries use, although it won't match exactly. For example, the real members names are uglified, and they may appear in a different order from what is shown here. But the design principles still apply.

[2] Assuming you're using a well-behaved allocator.

[3] The capacity never goes below `BUFSIZE - 1`.