Performance & Memory Considerations

It's time to see how string_view fares in terms of memory efficiency.

WE'LL COVER THE FOLLOWING

The core idea behind adding `string_view` into the Standard Library was performance and memory consumption. By leveraging `string_view`, you can efficiently skip the creation of many temporary strings which might boost performance.

Memory

`string_view` is usually implemented as `[ptr, len]` - one pointer and usually `size_t` to represent the possible size.

That's why you should see the size of it as 8 bytes or 16 bytes (depending on whether the architecture is x86 or x64).

If we consider the `std::string type`, due to common **Small String Optimizations**, `std::string` is usually 24 or 32 bytes, so double the size of `string_view`. If a string is longer than the SSO buffer then `std::string` allocates memory on the heap. If SSO is not supported (which is rare), then `std::string` would consist of a pointer to the allocated memory and the size.

Performance

`string_view` has only a subset of string operations, those that don't modify the referenced character sequence. Functions like `find()` should offer the same performance as the `string` counterparts.

On the other hand, `substr` is just a copy of two elements in `string_view`, while

`string` will perform a copy of a memory range. The complexity is `O(1)` vs `O(n)`. That's why if you need to split a larger string and work with those splices, the `string_view` implementation should offer better speed.

Let's look at strings in constant expressions.