`fixed_string`

why we need another string type and how to produce a never-dangling `string_view`

Dr. Matthias Kretz

GSI Helmholtz Centre for Heavy Ion Research

2024-10-30

@mkretz@floss.social

github.com/mattkretz

C strings

C Strings

- An array of $N + 1$ `char`s, where N is the length of the string.
- The last `char` in the string must be `'\0'`.
- Null characters anywhere else in the string not possible (terminate the string).
- `sizeof("foo") == 4`. Most efficient way to *store* length of the string.
- `char*` has an additional size overhead of `sizeof(char*)`

These are all equivalent:

```
1 const char* str1 = "foo";
2 const char str2[4] = {'f', 'o', 'o', '\0'};
3 using char4 = char[4];
4 const char4& str3 = "foo";
```

Problems

- Size information (null terminator) is easily overwritten/forgotten.
- Size needs to be computed `std::strlen`.

slow for long strings / text

- Lifetime of anything that's not a literal must be managed manually.

Take-Away #1

C strings are error prone and should be avoided.

std::string

`std::string` is a dynamically allocating string class, similar to a `std::vector<char>`

- It has a `capacity` and `size` (`capacity >= size`).
- Allocates memory on the heap.
- Requires three members internally: `begin`, `size`, `capacity`. (or pointers instead of `size`/`capacity`)

$\Rightarrow$ `sizeof(std::string) >= 3 * sizeof(void*)`.

```
1 std::string s = "foo";
```

- 4 Bytes for the string literal.
- ≥ 24 Bytes for the `std::string` object.
- *n* Bytes allocated on the heap?

SSO: small string optimization

small string optimization

If the string is “small enough“, reinterpret some local members as `char` array.

```
#include <string>

auto f()
{
  std::string s = "foo";
  return s;
}
```

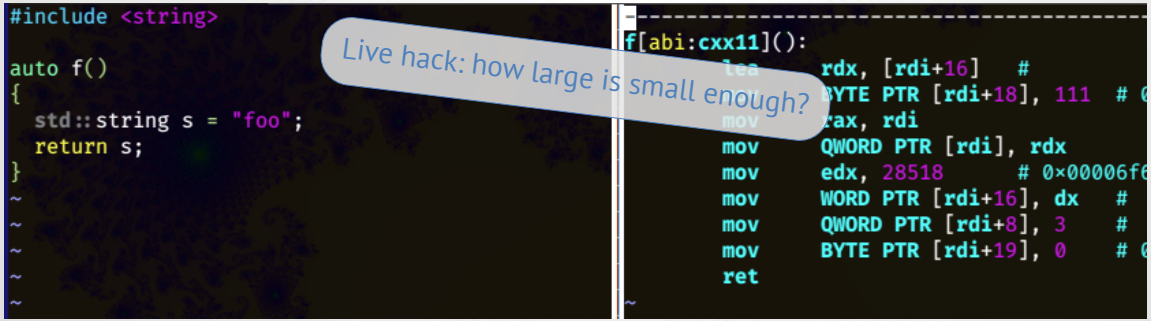
```
f[abi:cxx11]():
        lea     rdx, [rdi+16]   #
        mov     BYTE PTR [rdi+18], 111  # 0
        mov     rax, rdi
        mov     QWORD PTR [rdi], rdx
        mov     edx, 28518      # 0x00006f6
        mov     WORD PTR [rdi+16], dx   #
        mov     QWORD PTR [rdi+8], 3    #
        mov     BYTE PTR [rdi+19], 0    # 0
        ret
```

```
'o'== 111 and 'f'== 102 and ('o'<< 8) | 'f'== 28518
```

SSO: small string optimization

small string optimization

If the string is “small enough“, reinterpret some local members as `char` array.




```
'o'== 111 and 'f'== 102 and ('o'<< 8) | 'f'== 28518
```

Take-Away #2

`std::string`s with up to 15 chars don't require heap allocation with libstdc++.

The `span` for strings

A safer & faster C string (`const char*`) replacement:

- Pointer to the `char` array plus size
- `sizeof(std::string_view) == 2 * sizeof(void*)`
- Null terminator not used/needed. (⇒ no `c_str()` member function)
- May contain `'\0'` in the string.

```
1 std::string_view make_foo1() {
2   const char* str = "foo";
3   return std::string_view(str, 3);
4 }
```

fine

```
1 std::string_view make_foo2() {
2   const char str[] = "foo";
3   return std::string_view(str, 3);
4 }
```

UB: dangling `string_view`!

Take-Away #3

Use `std::string_view` instead of `const char*`.

NTTP

Ever tried to pass a string as template parameter?

NTTP

Non-type template parameter

```
1  template <auto X> void f(); { … }
2
3  void test() {
4    f<1>(); // OK
5
6    f<"foo">(); // Error
7    f<std::string("foo")>(); // Error
8    f<std::string_view("foo")>(); // Error
9
10   static const char str[] = "foo";
11   f<str>(); // OK
12 }
```

So it's possible.
It's just verbose and hard to use.

The C++ committee is going to see papers to make `std::string` and `std::string_view` valid NTTPs. So maybe in C++26 or C++29 …

NTTP

Ever tried to pass a string as template parameter?

NTTP

Non-type template parameter

```
1  template <auto X> void f(); { … }
2
3  void test() {
4    f<1>(); // OK
5
6    f<"foo">(); // Error
7    f<std::string("foo")>(); // Error
8    f<std::string_view("foo")>(); // Error
9
10   static const char str[] = "foo";
11   f<str>(); // OK
12 }
```

So it's possible.
It's just verbose and hard to use.

The C++ committee is going to see papers to make `std::string` and `std::string_view` valid NTTPs. So maybe in C++26 or C++29 …

NTTP

Ever tried to pass a string as template parameter?

NTTP

Non-type template parameter

```
1  template <auto X> void f(); { … }
2
3  void test() {
4    f<1>(); // OK
5
6    f<"foo">(); // Error
7    f<std::string("foo")>(); // Error
8    f<std::string_view("foo")>(); // Error
9
10   static const char str[] = "foo";
11   f<str>(); // OK
12 }
```

So it's possible.
It's just verbose and hard to use.

The C++ committee is going to see papers to make `std::string` and `std::string_view` valid NTTPs. So maybe in C++26 or C++29 …

Requirements

- valid NTTP ⇒ structural type ⇒ data members are public
- implicitly converts from string literal
- `string`-like functionality
- Usable as C string? (`c_str()` member function?)
- minimize size overhead — because we can

Example

… to the terminal

Take-Away #4

`fixed_string` acts a lot like a more intuitive string literal replacement

How to place a computed object into .rodata

- In a `consteval` function you might want to compute a new string.
- But a string in C++ is very often allocated memory + pointer to that memory.
- So if you need something else than `fixed_string` you need that pointer.
- Where is the memory?

Solution idea

References to constant expressions (and thus also template parameters!) materialize in `.rodata`.

How to place a computed object into .rodata

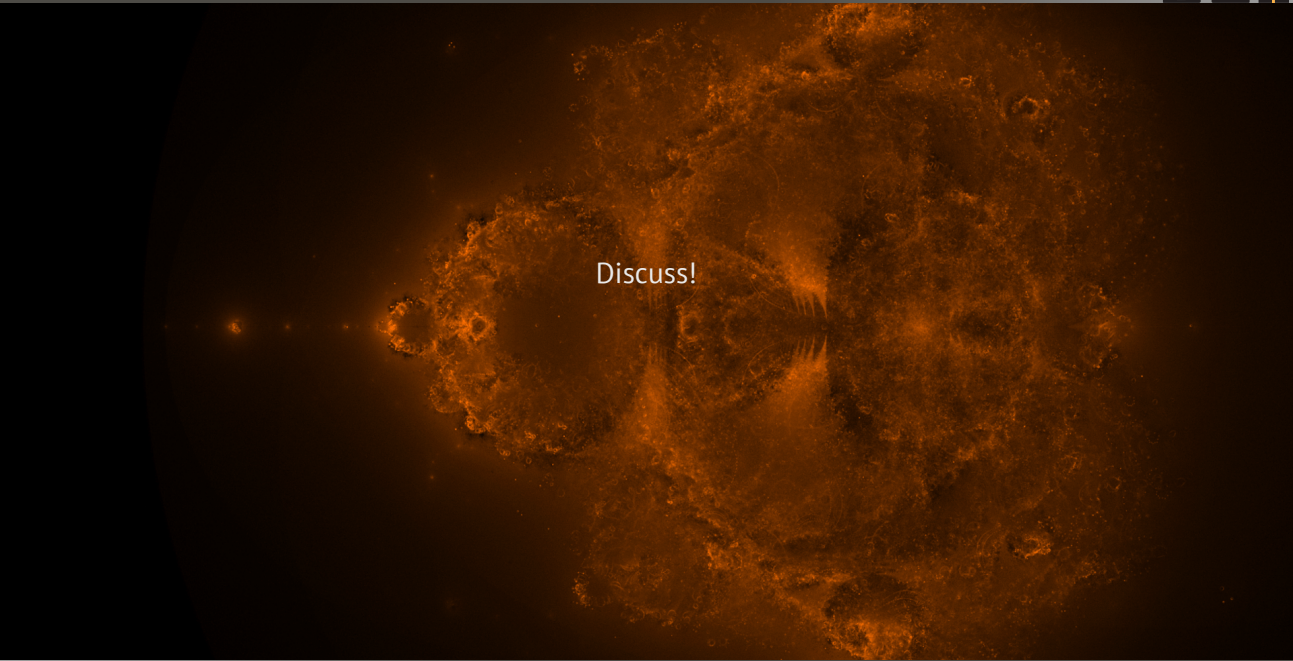
- In a `consteval` function you might want to compute a new string.
- But a string in C++ is very often allocated memory + pointer to that memory.
- So if you need something else than `fixed_string` you need that pointer.
- Where is the memory?

Solution idea

References to constant expressions (and thus also template parameters!) materialize in `.rodata`.

Take-Away #5

`constexpr_string` can be constructed to guard against dangling views


Discuss!