

Enough `string_view` to hang ourselves

September, 2018



Victor Ciura

Technical Lead, Advanced Installer

www.advancedinstaller.com

Abstract

Wouldn't it be nice if we had a standard C++ type to represent strings ?

Oh, wait... we do: `std::string`.

Wouldn't it be nice if we could use that standard type throughout our whole application/project ?

Well... we can't ! Unless we're writing a console app or a service. But, if we're writing an app with GUI or interacting with modern OS APIs, chances are that we'll need to deal with at least one other non-standard C++ string type. Depending on the platform and project, it may be `CString` from MFC or ATL, `Platform::String` from WinRT, `QString` from Qt, `wxString` from wxWidgets, etc. Oh, let's not forget our old friend `const char*`, better yet `const wchar_t*` for the C family of APIs...

So we ended up with two string types in our codebase. OK, that's manageable: we stick with `std::string` for all platform independent code and convert back-and-forth to the other `XString` when interacting with system APIs or GUI code. We'll make some unnecessary copies when crossing this bridge and we'll end up with some funny looking functions juggling two types of strings; but that's glue code, anyway... right?

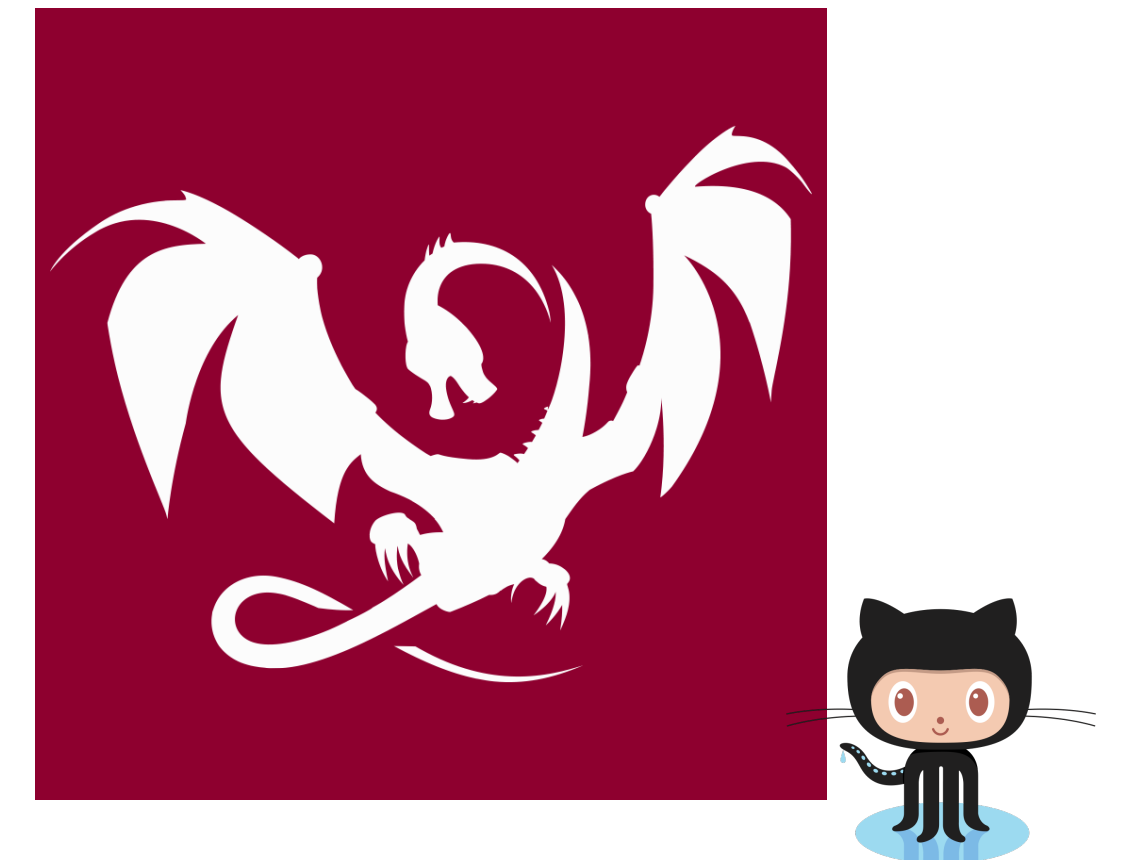
It's a good plan... until our project grows and we accumulate lots of string utilities and algorithms. Do we restrict those algorithmic goodies to `std::string` ? Do we fallback on the common denominator `const char*` and lose the type/memory safety of our C++ type ? Is C++17 `std::string_view` the answer to all our string problems ?

We'll try to explore our options, together...

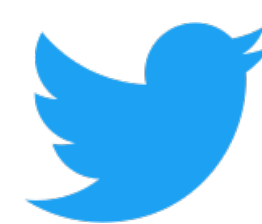
Who Am I?



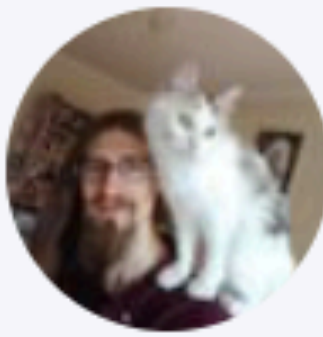
Advanced Installer



Clang Power Tools



@ciura_victor




Simon Brand @TartanLlama · Jul 6

So this happened

#CppCon





Thaaaaats

You are here ---->

Monday, September 24

11:00

Enough string_view to Hang Ourselves [Manage](#)
Victor Ciura

Tuesday, September 25

09:00

Regular Types and Why Do I Care ? [Manage](#)
Victor Ciura

Wednesday, September 26

09:00

These Aren't the COM Objects You're Looking For [Manage](#)
Victor Ciura

Thursday, September 27

09:00

Better Tools in Your Clang Toolbox: Extending clang-tidy With Your Custom Checks
Victor Ciura

Enough `string_view` to hang ourselves



Part 2 of N

Why `string_view` ?

Why are we talking about strings ?

Have we really exhausted all the cool C++ `template<>` topics 🤪 ?

ATL CString
MFC CString
CComBSTR
wxString
QString
BSTR
winrt::hstring
Platform::String
XString
WTF::String
WTF::CString
const char*

 WTF: **W**eb **T**emplate **F**ramework (WebKit)



SSO



CoW

thread-safe ?

char traits ?

custom allocator ?



String literals ?



`constexpr`

Encodings ?

 **Unicode ?**

 **Locale ?**

Forget it !



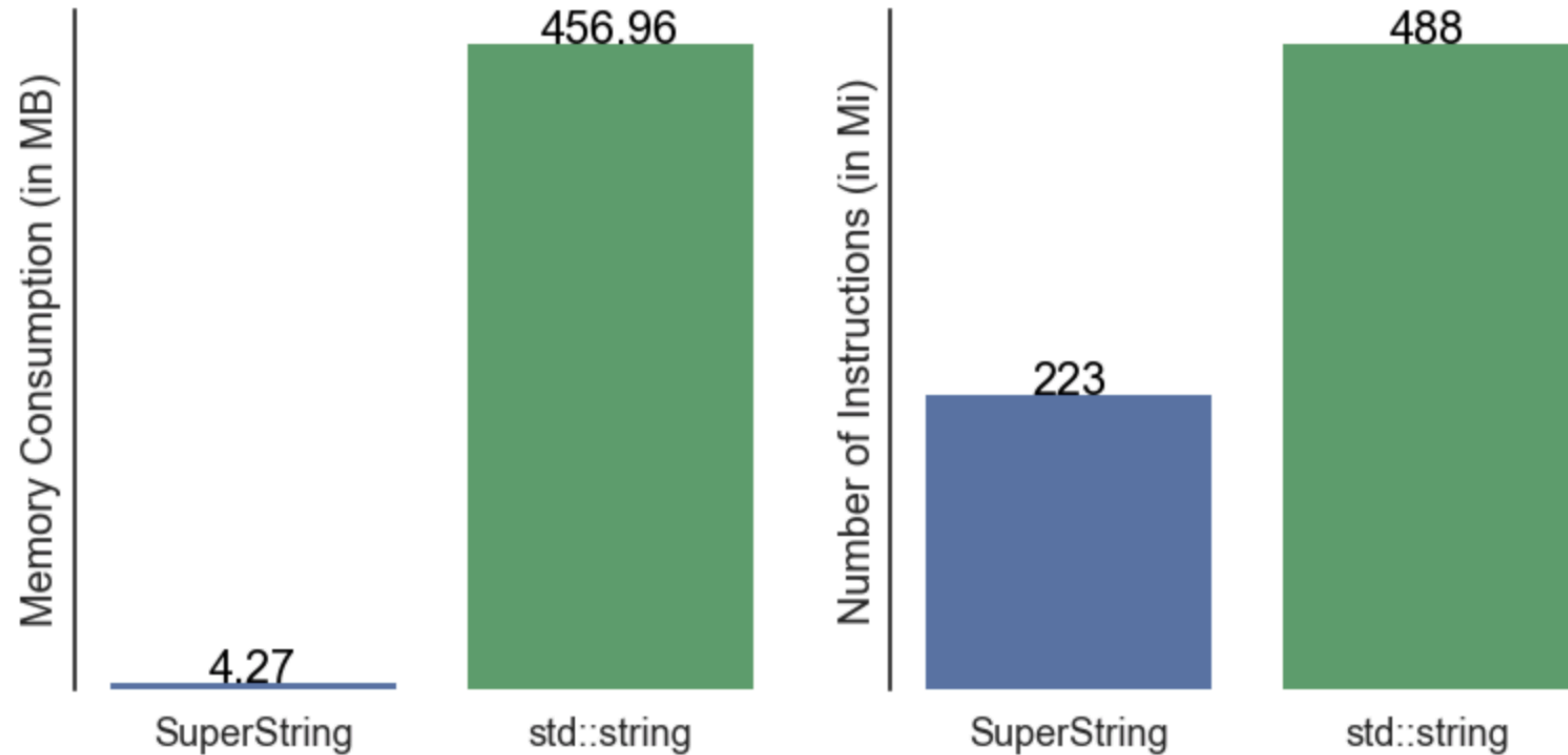
C++ Super String to the rescue !



SuperString

SuperString is an efficient string library for C++, that achieves a remarkable memory and CPU optimization.

SuperString uses Rope (data structure) and game theory techniques.




<https://www.boutglay.com/SuperString/>

<https://github.com/btwael/SuperString>



So we ended up with something like this...

XString ↔  `const char*` ↔ `std::string`
`const char[N]`

+ glue code





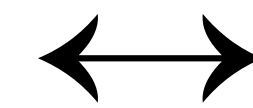
String Algorithms

XString



const char*

const char[N]



std::string



Is C++17 `string_view`
the answer to all our string problems ?

We'll try to explore our options, together...

On more serious note...

C++98/03

C++11

C++14

C++17



No C++17, no problem...

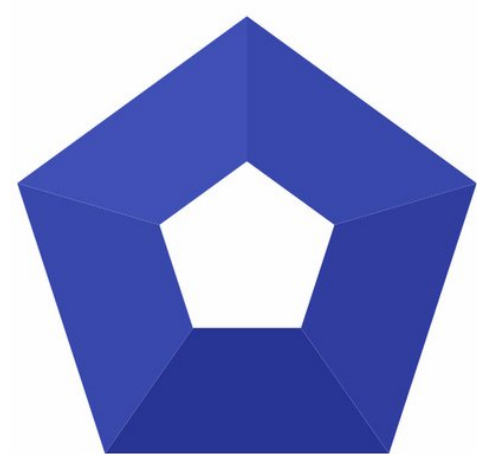
`absl::string_view`

`StringPiece` (Google)

`StringRef` (LLVM)

`boost::string_ref`

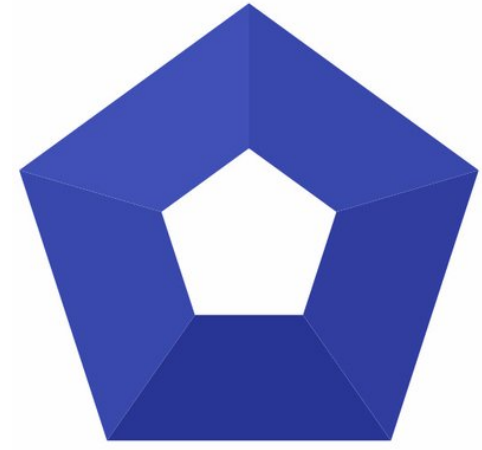
`folly::Range`



`absl::string_view`
(Google)

- **Rationale**
- **Exemples**
- **Gotchas**

https://abseil.io/docs/cpp/guides/strings#string_view

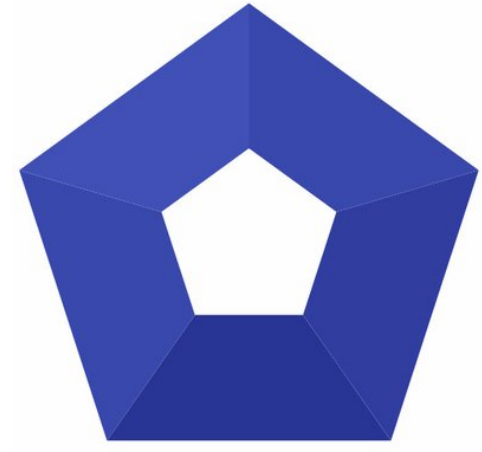


Abseil

(Google)

Abseil is an **open-source** collection of C++ library code designed to augment the C++ standard library.

The Abseil library code is collected from Google's own C++ code base, has been extensively tested and used in production.

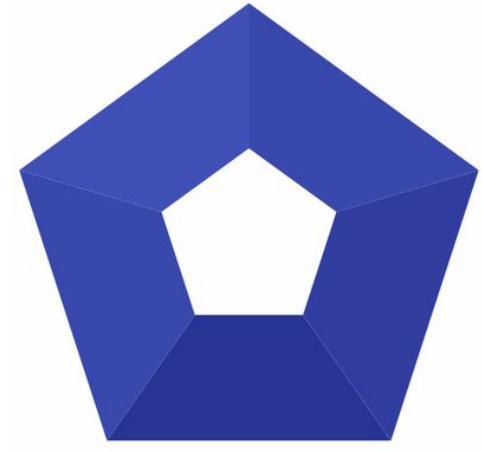


Abseil

(Google)

Abseil is not meant to be *competitor* to any standard library code.

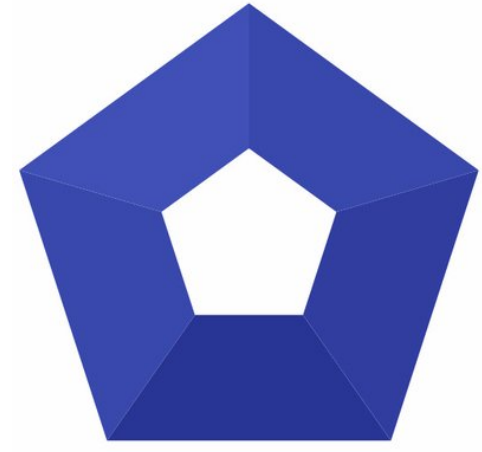
Many of these utilities serve a purpose within Google's code base, and they now want to provide those resources to the C++ community as a whole.



Abseil
(Google)

In some cases, Abseil provides pieces missing from the C++ standard.

In others, Abseil provides *alternatives* to the standard for special needs



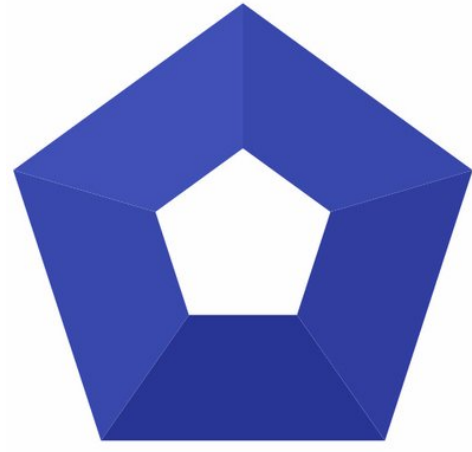
Abseil
(Google)

Do you remember **Herb Sutter's** **GotW** series ?

<https://herbsutter.com/gotw/>

Abseil C++ Tips of the Week

<https://abseil.io/tips/>



Abseil C++ Tips of the Week

Tip of the Week #1: `string_view`

<https://abseil.io/tips/1>

cppreference.com Create account

Page Discussion View Edit History

C++ [Strings library](#) **std::basic_string_view**

std::basic_string_view

Defined in header `<string_view>`

```
template<
    class CharT,
    class Traits = std::char_traits<CharT> (since C++17)
> class basic_string_view;
```

The class template `basic_string_view` describes an object that can refer to a constant contiguous sequence of `char`-like objects with the first element of the sequence at position zero.

A typical implementation holds only two members: a pointer to constant `CharT` and a size.

Several typedefs for common character types are provided:

Defined in header `<string_view>`

Type	Definition
<code>std::string_view</code>	<code>std::basic_string_view<char></code>
<code>std::wstring_view</code>	<code>std::basic_string_view<wchar_t></code>
<code>std::u16string_view</code>	<code>std::basic_string_view<char16_t></code>
<code>std::u32string_view</code>	<code>std::basic_string_view<char32_t></code>

Template parameters

CharT - character type

Traits - `CharTraits` class specifying the operations on the character type. Like for `basic_string`, `Traits::char_type` must name the same type as `CharT` or the behavior is undefined.

`std::string_view`

“The class template `basic_string_view` describes an object that can refer to a **constant** *contiguous* sequence of `char`-like objects.”

std::string_view

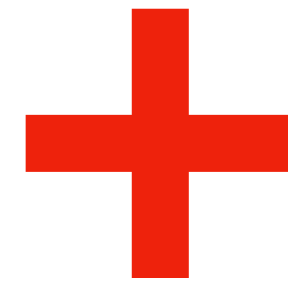
A typical *implementation* holds only two members:

- a *pointer* to constant CharT
- a *size*

`std::string_view`

A `string_view` has almost exactly the same interface as a **constant** `std::string`

std::string_view **Modifiers**



```
constexpr void remove_prefix(size_type n);
```

```
constexpr void remove_suffix(size_type n);
```

std::string_view

A *lightweight* string-like view into an array of characters.

It can be constructed from a `const char*` (null terminated) or from a *pointer and a length*.

Intended to be used in combination with ***string literals***.

std::string_view

Design goal: avoid temporary `std::string` objects.

`std::string_view` was designed to interoperate with `std::string` 😈

Caveat: comes with some *usage complexity* (gotchas).

`std::string_view`

Why are we trying to avoid *temporary* `std::string` objects ?

- they're not small (3 pointers)
- they can cause memory allocation
- **SSO** mitigates this... somewhat
- you have to copy the data around (unnecessarily) -- **CoW** ?

`std::string_view`

A `string_view` does not manage the **storage** that it refers to.

Lifetime management is up to the user (caller).

`std::string_view`

When would you use a `std::string_view` instead of a `std::string` ?

We'll try to explore our options, together...

std::string_view

```
constexpr basic_string_view() noexcept;
```

```
constexpr basic_string_view(const basic_string_view & other) noexcept;
```

```
constexpr basic_string_view(const CharT * s, size_type count);
```

```
constexpr basic_string_view(const CharT * s); 🙌
```


std::string_view

The costly constructor:

```
constexpr basic_string_view(const CharT * s);
```

Must *calculate* the **length** of the passed null terminated string.

Which might happen at compile-time or might not.

std::string

```
operator std::basic_string_view<CharT, Traits>() const noexcept;
```

Returns a `std::basic_string_view`

constructed as: `{ data(), size() }`

Convenience Conversions (and Gotchas)

- `const char *` automatically converts to `std::string` via constructor (*not explicit*)
- `const char *` automatically converts to `std::string_view` via constructor (*not explicit*)
- `std::string` automatically converts to `std::string_view` via *conversion operator*
- can construct a `std::string` from a `std::string_view` via constructor (*explicit*)

Storing string_views

TL;DR; Don't !

Storing string_views in a container is potentially risky.
You can end up holding onto freed memory (temporary strings).

```
std::vector<std::string_view> elements;
```

```
void Save(const std::string & elem)
{
    elements.push_back(elem); // automatic conversion: ouch!
}
```



no compiler warning

Containers and string_views

How about the other way around ?

```
std::map<std::string, int> frequencies;
```

```
int GetFreqForKeyword(std::string_view keyword)
{
    return frequencies.at(keyword);
}
```


Containers and string_views

How about the other way around ?

```
std::map<std::string, int> frequencies;
```

```
int GetFreqForKeyword(std::string_view keyword)
{
    return frequencies.at(keyword);
}
```



doesn't compile: no conversion to `std::string`

Transparent Comparators

```
std::map<std::string, int, less<>> frequencies;
```

 `std::less<>` provides `is_transparent`

```
int GetFreqForKeyword(std::string_view keyword)
{
    if (auto it = frequencies.find(keyword); it != frequencies.end())
        return *it;

    return 0;
}
```

Transparent Comparators

```
std::map<std::string, int, less<>> frequencies;
```

 `std::less<>` provides `is_transparent`

```
int GetFreqForKeyword(std::string_view keyword)
```

```
{
```

```
    if (auto it = frequencies.find(keyword); it != frequencies.end())  
        return *it;
```



```
operator< (std::string_view, std::string_view)  
          (in the box)
```

```
    return 0;
```

```
}
```

Don't worry, your `compiler` will help you

Well...

string and string_view

```
class Sink  
{  
public:
```

```
    Sink(std::string_view sv) : str(std::move(sv)) {}
```

```
private:
```

```
    std::string str;  
};
```



Does this compile ?

YES

std::string

Special Constructor

```
template < class T >  
explicit basic_string(const T & t, const Allocator & alloc = Allocator());
```

IF `t` is convertible to `std::basic_string_view` to initialize the string with.

Implicitly converts `t` to a string view `sv` as if by:

```
basic_string_view<CharT, Traits> sv = t;
```

then initializes the string with the contents of `sv` , as if by:

```
basic_string(sv.data(), sv.size(), alloc)
```

string and string_view

```
class Sink  
{  
public:
```

```
    Sink(std::string_view sv) : str(std::move(sv)) {}
```

```
private:
```

```
    std::string str;  
};
```

 **Don't do this !**

string and string_view

 **Don't use `std::string_view` to initialize a `std::string` member !**

If you know that you're ultimately going to create a `std::string`
make the whole **call chain** use `std::string`
don't **mix-in** `std::string_view` along the way.

 `string_view` → `string` → `string_view` → ... → `std::string`
(*maybe a string literal*)

But surely `static analysis tools` have checks for strings...

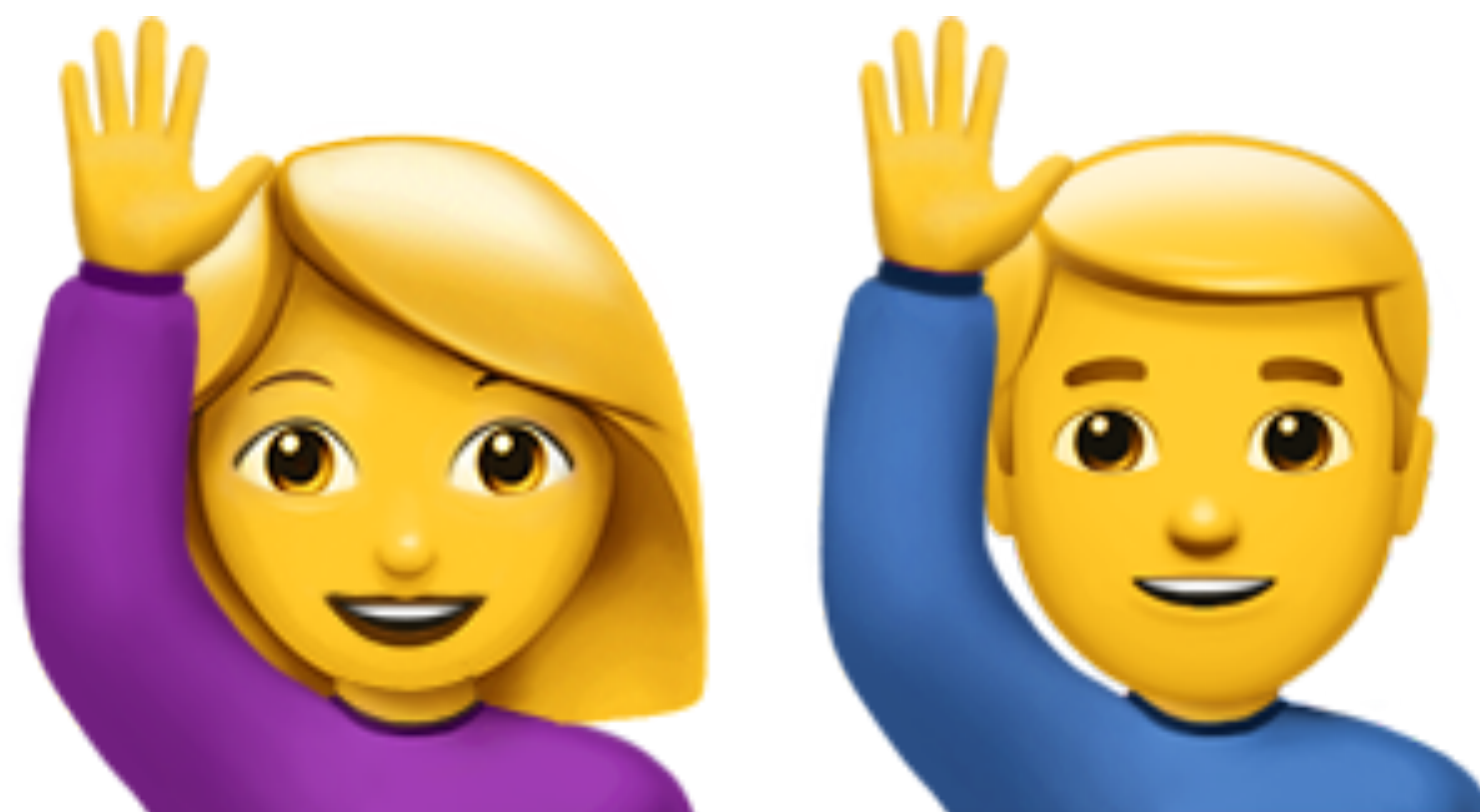


Let's see...



clang-tidy

Who used clang-tidy ?





+



->



LLVM

Visual Studio

Clang Power Tools

clang-tidy

2015/2017

www.clangpowertools.com

clang-format

FREE / Open source

Clang-Tidy Checks

- **abseil-string-find-startswith**
 - Options
- **android-cloexec-accept**
- **android-cloexec-accept4**
- **android-cloexec-creat**
- **android-cloexec-dup**
- **android-cloexec-epoll-create**
- **android-cloexec-epoll-create1**
- **android-cloexec-fopen**
- **android-cloexec-inotify-init**
- **android-cloexec-inotify-init1**
- **android-cloexec-memfd-create**
- **android-cloexec-open**
- **android-cloexec-socket**
- **android-comparison-in-temp-failure-retry**
- **boost-use-to-string**
- **bugprone-argument-comment**
 - Options
- **bugprone-assert-side-effect**
 - Options
- **bugprone-bool-pointer-implicit-conversion**
- **bugprone-copy-constructor-init**



clang-tidy

over 250 checks

<https://clang.llvm.org/extra/clang-tidy/checks/list.html>

String related checks



clang-tidy

- abseil-string-find-startswith
- boost-use-to-string
- bugprone-string-constructor
- bugprone-string-integer-assignment
- bugprone-string-literal-with-embedded-nul
- bugprone-suspicious-string-compare
- modernize-raw-string-literal
- performance-faster-string-find
- performance-inefficient-string-concatenation
- readability-redundant-string-cstr
- readability-redundant-string-init
- readability-string-compare

<https://clang.llvm.org/extra/clang-tidy/checks/list.html>

String related checks



clang-tidy

No check for `string_view`



<https://clang.llvm.org/extra/clang-tidy/checks/list.html>

String related checks



Oh, wait !

clang-tidy

bugprone-dangling-handle 🤔

”

Detect dangling references in value handles like `std::string_view`.

These dangling references can be a result of constructing handles from temporary values, where the temporary is destroyed soon after the handle is created.

<https://clang.llvm.org/extra/clang-tidy/checks/bugprone-dangling-handle.html>



Lifetime profile v1.0

<https://herbsutter.com/2018/09/20/lifetime-profile-v1-0-posted/>

- “ A dangling `string_view`, which is important because it turns out to be **easy** to convert a `std::string` to a `string_view`, so that **dangling is almost the default behavior**.



CppCoreGuidelines

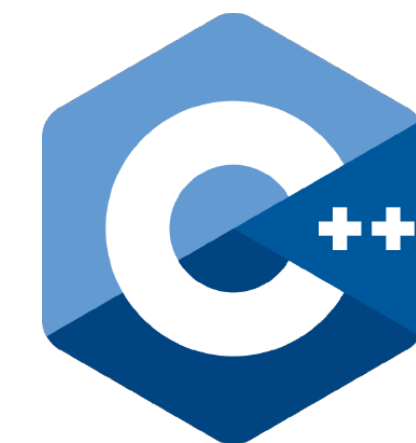
<https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf>



Lifetime profile v1.0

<https://herbsutter.com/2018/09/20/lifetime-profile-v1-0-posted/>

```
void example_2_6_2_1()
{
    std::string_view s = "foo"s;    // A
    s[0];    // ERROR (Lifetime.3): 's' was invalidated when
            // temporary "foo"s' was destroyed (line A)
}
```



CppCoreGuidelines

<https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf>



Lifetime profile v1.0

<https://herbsutter.com/2018/09/20/lifetime-profile-v1-0-posted/>

```
<source>:7:5: warning: passing a dangling pointer as argument [-Wlifetime]
    s[0];           // ERROR (lifetime.3): 's' was invalidated when
    ^
<source>:6:32: note: temporary was destroyed at the end of the full expression
    std::string_view s = "foo"s;           // A
                          ^
1 warning generated.
Compiler returned: 0
```

clang **-Wlifetime**



CppCoreGuidelines

<https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf>

std::string_view cheatsheet

Lifetime with std::string_view (C++17)

std::string_view isn't a drop-in replacement
for const std::string&

```
std::string str() {  
    return std::string("long_string_helps_to_detect_issues");  
}
```

```
const std::string& s = str();  
std::cout << s << '\n';
```

lifetime extended
prints the correct result



```
std::string_view sv = str();  
std::cout << sv << '\n';
```

lifetime not extended
prints nonsense



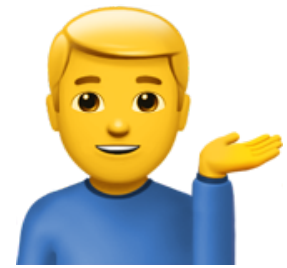
const lvalue reference binds to rvalue and provides lifetime extension. But there is no lifetime extension for std::string_view.

For short strings this issue might be hard to detect due to short string optimization (SSO). The problem becomes obvious with longer (dynamically allocated) strings.

@walletfox



Reference Lifetime Extension



You shouldn't be relying on *reference lifetime extension* in the first place

Really try to understand **copies**
(copy-ellipsis, RVO, SSO)

`std::literals::string_view_literals::operator""sv`

```
using namespace std::literals;
```

```
std::string_view s1 = "abc\0\0def";  
std::string_view s2 = "abc\0\0def"sv;
```

```
std::cout << "s1: " << s1.size() << " '" << s1 << "'";  
std::cout << "s2: " << s2.size() << " '" << s2 << "'";
```

What does this print ?

GODBOLT ALL THE THINGS !!!



<https://godbolt.org>



`std::literals::string_view_literals::operator""sv`

```
using namespace std::literals;
```

```
std::string_view s1 = "abc\0\0def";  
std::string_view s2 = "abc\0\0def"sv;
```

```
std::cout << "s1: " << s1.size() << " '" << s1 << "'";  
std::cout << "s2: " << s2.size() << " '" << s2 << "'";
```

<https://godbolt.org/z/pMBXDR>

What if I want a NULL terminated string ?

`c_str()`

```
template <typename T>
auto c_str(const std::basic_string_view<T> & view) noexcept
{
    if ( *(view.data() + view.size()) )
    {
        std::terminate();
    }

    return view.data();
}
```

**So I was determined that I wanted some
string_view in my life...**



How to Return?

```
template<typename T>
void print (??? x) {
    std::cout << typeid(x).name() << '\n';
}

void print (T x); //vc e c c ve

void print (const T& x); //c c c c c

template<typename T>
??? dbl (??? x) {
    return x + x;
}

T dbl (T x); //vc e dnc dnc FATAL c
T dbl (const T& x); //c c dnc dnc FATAL c
```

```
int i = 42;
auto r = dbl(i); //cheap return

std::string s = "pretty long string";
auto r = dbl(s); //cheap return with RVO

std::string s = "pretty long string";
auto r = dbl(std::ref(s)); //Compile-Time Error
auto r = dbl(std::cref(s)); //Compile-Time Error

print("hello"); //Compile-Time Error

string operator+(string_view, string_view);
std::string_view sv = "long stringview";
auto r = dbl(s); //OOPS
std::cout << r << '\n'; //FATAL Run-Time Error
```

43

josuttis | eckstein
IT communication



C++ Templates Revised

43:52 / 1:00:40

CppCon.org

CppCon 2017: Nicolai Josuttis "C++ Templates Revised" https://www.youtube.com/watch?v=ULX_VTkMvf8

Nicolai Josuttis:

"string_view is a nightmare !"

"string_view is worse than a string&"

"string_view is *semantically* a string&
but *conceptually* a value type"

Me



Nicolai Josuttis:

```
template<typename T>  
T dbl(T x)  
{  
    return x + x;  
}
```

```
string operator+ (string_view s1, string_view s2);
```

```
string_view sv = "long string view";
```

```
auto r = dbl(sv);    // Outch !  
std::cout << r << "\n";    // runtime error
```


Nicolai Josuttis:

"Don't assign to auto"

"Almost Always Auto (AAA) is broken with string_view"

`std::string_view` Considered Harmful

- **Worse than `std::string&`**
 - prvalue/reference semantics with lvalue
 - but no lifetime extension
- **Don't use `std::string_view` to initialize a `std::string`**
 - No `std::string` at the end of a `std::string_view` chain
 - No `std::string_view` as arg in constructor for `std::string` member

(Thanks to Jason Turner)
- **Never return `std::string_view`**
- **Never initialize a returned value as `std::string_view`**
- **Function templates should never return argument type `T`**
 - Return **auto**
- **Don't assign to auto**
 - **AlmostAlwaysAuto** is broken with `std::string_view`

Nicolai Josuttis:



```
template<typename T>  
auto dbl(T x)  
{  
    return x + x;  
}
```



```
string operator+ (string_view s1, string_view s2);
```

```
string_view sv = "long string view";
```

```
auto r = dbl(sv);    // now r is a std::string  
std::cout << r << "\n";
```

Bind Returned/Initialized Objects to the Lifetime of Parameters

wg21.link/p0936

Richard Smith, Nico Josuttis: P0936R0 Bind Returned/Initialized Objects to Lifetime of Parameters



Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0936R0**
Date: 2018-02-12
Reply to: Richard Smith (richardsmith@google.com), Nicolai Josuttis (nico@josuttis.de)
Audience: EWG, CWG
Prev. Version:

Bind Returned/Initialized Objects to the Lifetime of Parameters, Rev0

Lifetime issues with references to temporaries can lead to fatal and subtle runtime errors. This applies to both:

- Returned references (for example, when using strings or maps) and
- Returned objects that do not have value semantics (for example using `std::string_view`).

This paper proposes a new language feature to detect and avoid these errors.

Motivation

Let's motivate the feature for both classes not having value semantics and references.

Classes not Having Value Semantics

C++ allows the definition of classes that do not have value semantics. One pretty new but already famous example is `std::string_view`: The lifetime of a `string_view` object is bound to an underlying string or character sequence.

Because string has an implicit conversion to `string_view`, it is easy to accidentally program a `string_view` to a character sequence that doesn't exist anymore.

WIP.. C++20

By the way...

`string_view (nullptr)`

ISO, Rapperswil, 2018

There was a *long* discussion in a joint EWG/LEWG session about the seemingly trivial issue of whether to change `string_view(nullptr)` from **undefined behavior** to constructing an **empty string_view**.

The result was to *keep the status quo*: **UB**

A few more months passed...



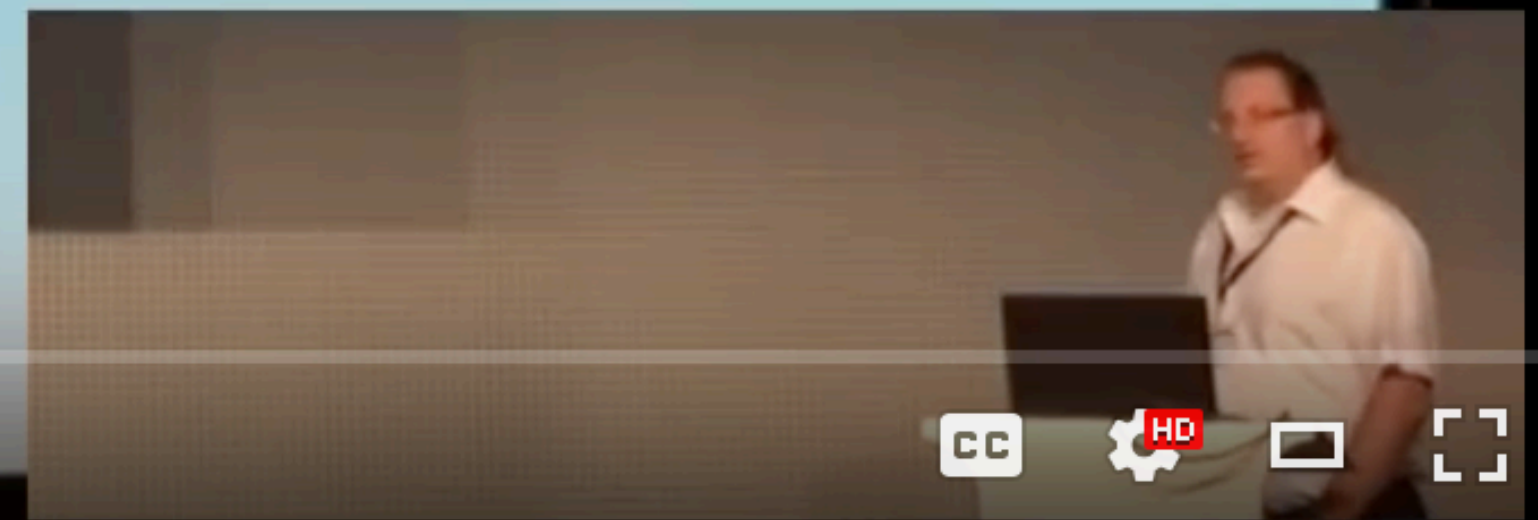
Meeting C++ 2017

Marc Mutz

Stringviews, Stringviews Everywhere

*StringViews,
StringViews everywhere!*

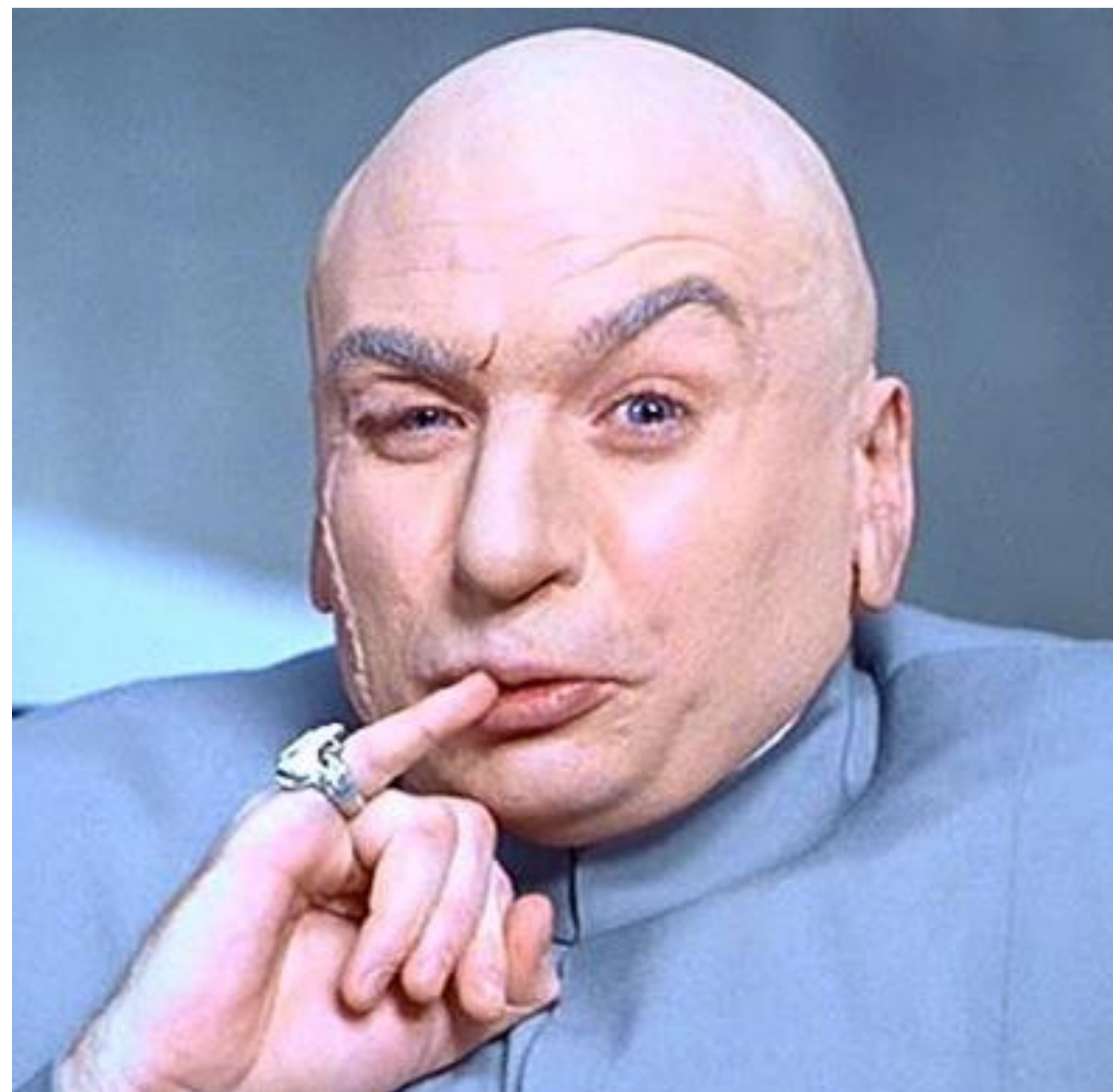
Marc Mutz <marc.mutz@kdab.com>



StringViews, StringViews everywhere! - Marc Mutz - Meeting C++ 2017

<https://www.youtube.com/watch?v=0QFPKgvLhao>

Me



I think I can do this...



Others there to help...



Jonathan Müller

`std::string_view` accepting temporaries: good idea or horrible pitfall ?

https://foonathan.net/blog/2017/03/22/string_view-temporary.html

Guidelines For Rvalue References In APIs

<https://foonathan.net/blog/2018/03/26/rvalue-references-api-guidelines.html>

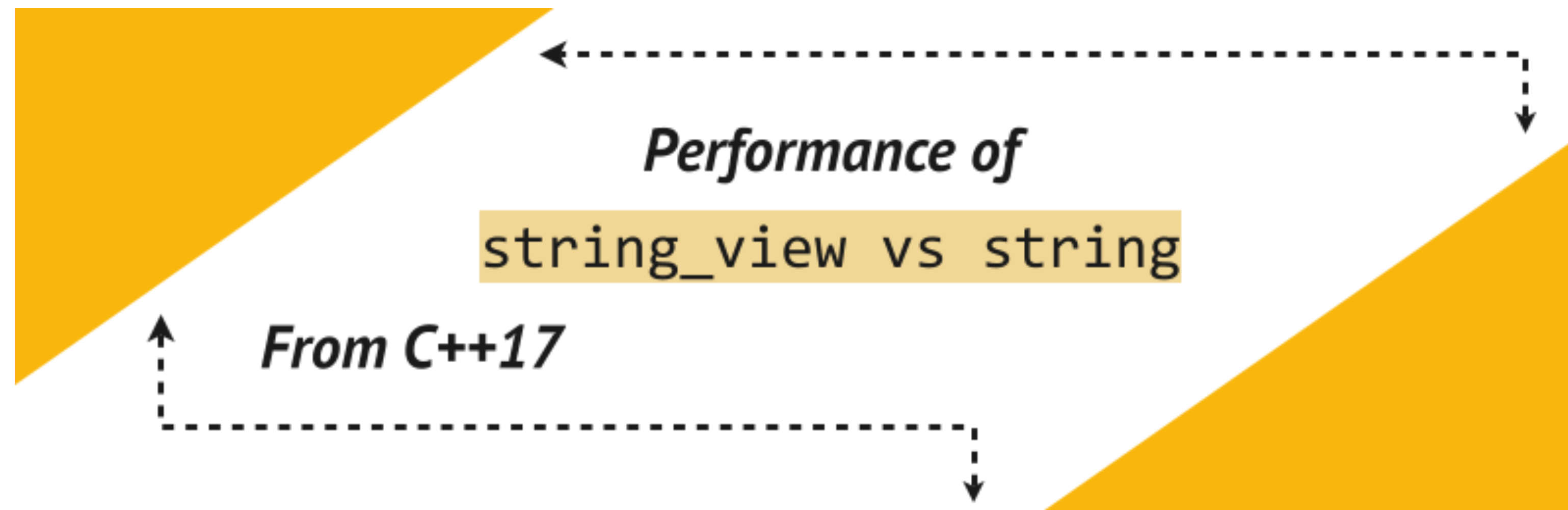
Andrzej Krzemiński's C++ blog

String's competing constructors

<https://akrzemi1.wordpress.com/2018/03/12/strings-competing-constructors/>

Bartek's Coding Blog

Performance of `std::string_view` vs `std::string`



Bartłomiej Filipek
@fenbf

<https://www.bfilipek.com/2018/07/string-view-perf.html>

Visual C++ blog

`std::string_view` : The Duct Tape of String Types

https://blogs.msdn.microsoft.com/vcblog/2018/08/21/stdstring_view-the-duct-tape-of-string-types/



Billy O'Neal
@MalwareMinigun



Arthur O'Dwyer

Stuff mostly about C++

`std::string_view` is a *borrow type*

<https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/>

`std::string_view` is a borrow type



`string_view` succeeds admirably in the goal of “*drop-in replacement*” for `const string&` parameters.

The problem:

The two relatively **old** kinds of types are **object types** and **value types**.

The new kid on the block is the ***borrow type***.

<https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/>

`std::string_view` is a borrow type

Borrow types are essentially “*borrowed*” references to existing objects.

- they lack ownership
- they are *short-lived*
- they generally can do without an *assignment operator*
- they generally appear only in *function parameter* lists
- they generally *cannot be stored in data structures or returned* safely from functions (no ownership semantics)

<https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/>

Non-owning reference types like `string_view` or `span`

You need more *contextual* information when working
on an instance of this type

Things to consider:

- shallow copy
- shallow/deep compare
- const/mutability
- `operator==`

`std::string_view` is a borrow type

`string_view` is perhaps the first “mainstream” *borrow type*.

BUT:

`string_view` is *assignable*: `sv1 = sv2`

Assignment has *shallow* semantics (of course, the viewed strings are *immutable*).

Meanwhile, the comparison `sv1 == sv2` has *deep* semantics.

<https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/>

std::string_view

Non-owning reference type

When the underlying data is extant and **constant** we can determine whether the rest of its usage still *looks Regular*

Generally, when used properly (as function parameter), `string_view` works well..., **as if** a **Regular** type

General Problem With Reference Types

For non-owning reference types like `string_view` or `span`

`operator==`

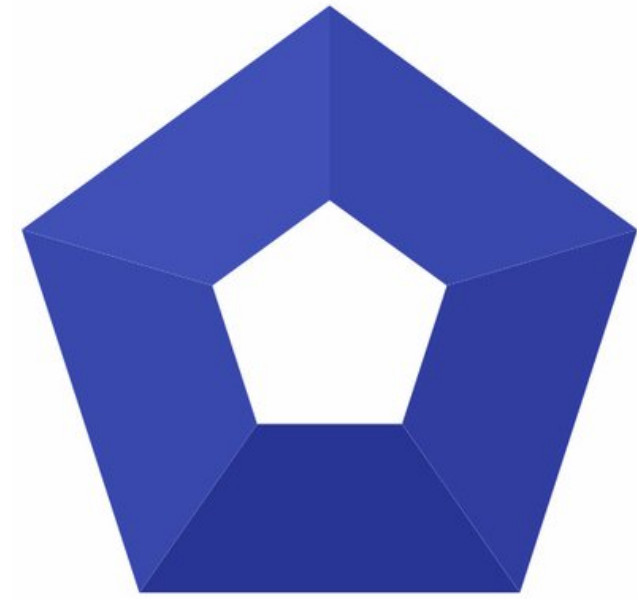
Try to restrict these types to **SemiRegular**
to avoid confusion for your users

C++20 `std::span<T>`

Think "array_view" as in `std::string_view`,
but **mutable** on underlying data



<https://en.cppreference.com/w/cpp/container/span>



Revisiting Regular Types (after 20 years)

<https://abseil.io/blog/20180531-regular-types>

Titus Winters, 2018

This essay is both the best up to date synthesis of the original **Stepanov** paper, as well as an investigation on using *non-values* **as if** they were **Regular** types.

This analysis provides us some basis to evaluate *non-owning reference parameters types* (like **string_view** and **span**) in a practical fashion, without discarding **Regular** design.

std::string_view Tries To Be Regular

***Non-Ownership and Generic Programming
and Regular types, oh my!***

<https://medium.com/@barryrevzin/non-ownership-and-generic-programming-and-regular-types-oh-my>

Barry Revzin
@barryrevzin

I have a whole talk just on that...

Regular Types and Why Do I Care ?

CppCon 2018

<http://sched.co/FnLB>

Simple rules for borrow types

Borrow types must appear *only as function parameters* and *for-loop control variables*.

We can make an **exception** for function *return types*:

- a function may have a borrow type as its return type
(the function must be **explicitly annotated** as returning a potentially dangling reference)
- the result returned ***must not be stored*** into any named variable, except a function parameter or for-loop control variable

<https://quuxplusone.github.io/blog/2018/03/27/string-view-is-a-borrow-type/>

Say What You Mean

If you decide to make an **exception** to these best practices, strongly consider **explicitly annotating** your intent in code.

Custom attributes ?



`[[magic]]`

`[[trust_me_on_this_one]]`

`[[i_am_very_sorry]]`

`[[it_works_on_my_machine]]`

`[[beware_of_dangling_reference]]`

<https://en.cppreference.com/w/cpp/language/attributes>

Credit: **Ólafur Waage**
@olafurw

I think **Marco Arena** said it best:

string_view odi et amo

https://marcoarena.wordpress.com/2017/01/03/string_view-odi-et-amo/

C++ Slack is your friend



<https://cpplang.slack.com>

CppLang Slack auto-invite:

<https://cpplang.now.sh/>



Cpplang

cpplang.slack.com



CppCast

```
auto CppCast = pod_cast<C++>("http://cppcast.com");
```



Rob Irving

@robwirving

Jason Turner

@lefticus

<http://cpp.chat>

<https://www.youtube.com/channel/UCsefcSZGxO9ITBqFbsV3sJg/>

<https://overcast.fm/itunes1378325120/cpp-chat>



Jon Kalb

@_JonKalb

Phil Nash

@phil_nash

Enough `string_view` to hang ourselves

September, 2018



[@ciura_victor](https://twitter.com/ciura_victor)

Victor Ciura

Technical Lead, Advanced Installer

www.advancedinstaller.com

Questions



[@ciura_victor](https://twitter.com/ciura_victor)