NDC { Oslo }

Enough `string_view` to hang ourselves

June 14, 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…

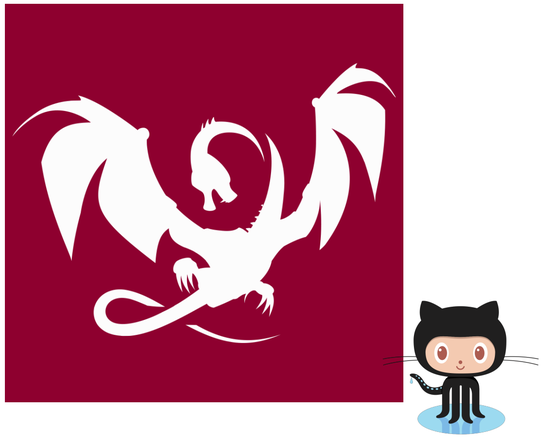
So we ended up with two string types in our codebase. OK, that's manageable: we stick with `std::string` for all platform independent code and convert back-and-forth to the other `XString` when interacting with system APIs or GUI code. We'll make some unnecessary copies when crossing this bridge and 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

Enough `string_view` to hang ourselves

 Part 1 of N

Why `string_view` ?

Why are we talking about `strings` ?

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

ATL CString

const wchar_t*

MFC CString

Platform::String

std::string

XString

wxString

WTF::String

WTF::CString

folly::fbstring

QString

const char*

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

thread-safe ?

📏 SSO

char traits ?

 CoW

custom allocator ?

🔡 String literals ?

 constexpr

Encodings ?

💪 Unicode ?

🇪🇺 Locale ?

Forget it !

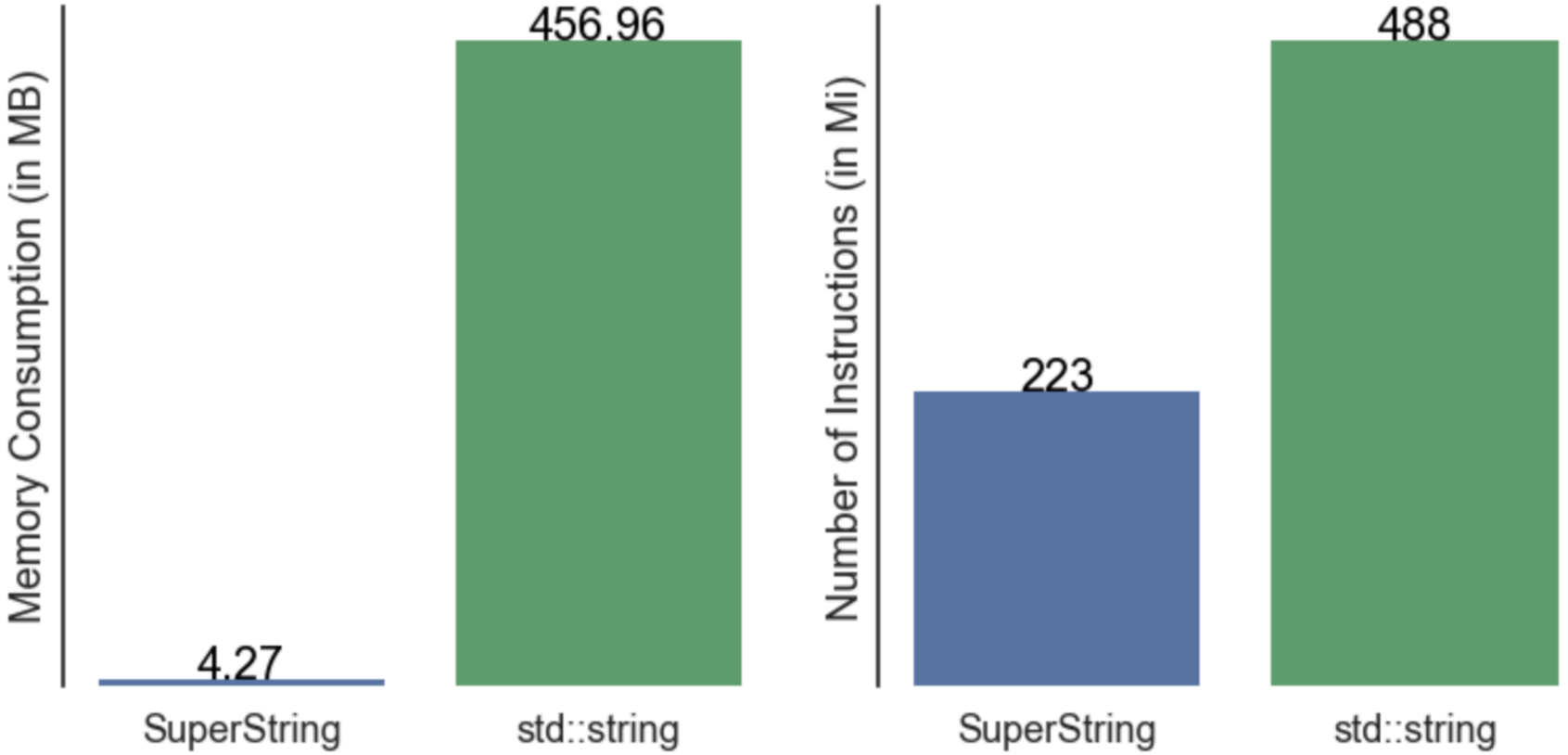
😞

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*` / `const char[N]` ⟷ `std::string`

+ glue code

String Algorithms

XString ⟷  const char* ⟷ std::string

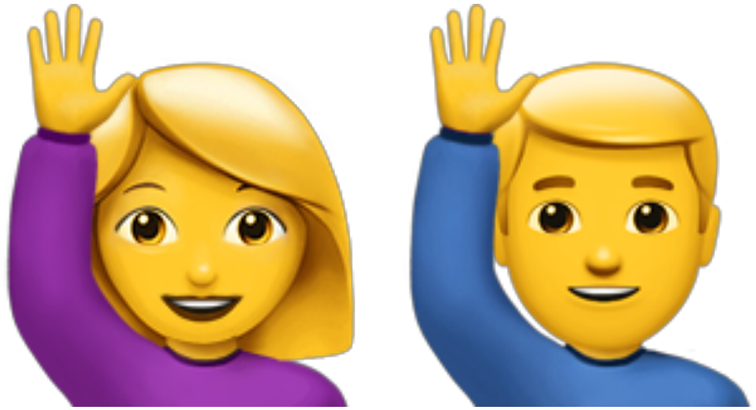
const char[N]

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...

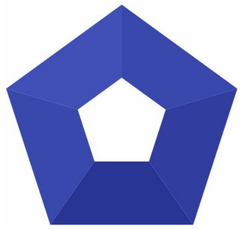
`absl::string_view`

`StringPiece` (Google)

`StringRef` (LLVM)

`boost::string_ref`

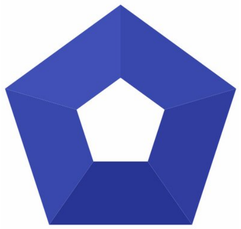
`folly::Range`



`absl::string_view` (Google)

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

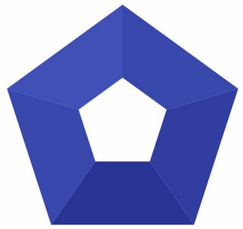
https://abseil.io/docs/cpp/guides/strings#string_view



Abseil
(Google)

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

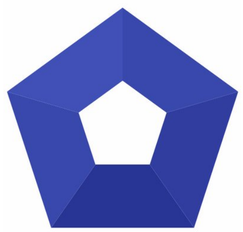
The Abseil library code is collected from Google's own C++ code base, has been extensively tested and used in production.



Abseil
(Google)

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

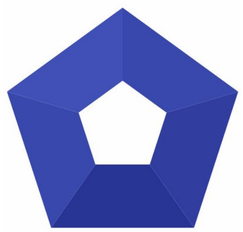
Many of these utilities serve a purpose within Google's code base, and they now want to provide those resources to the C++ community as a whole.



Abseil
(Google)

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

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



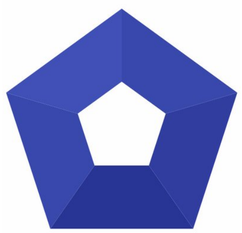
Abseil
(Google)

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

#include <string_view>

cppreference.com Create account Search

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>
> class basic_string_view;
```
(since C++17)

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
std::string_view	`std::basic_string_view<char>`
std::wstring_view	`std::basic_string_view<wchar_t>`
std::u16string_view	`std::basic_string_view<char16_t>`
std::u32string_view	`std::basic_string_view<char32_t>`

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;

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

  return 0;
}
```

👆 std::less<> provides is_transparent

👆

operator< (std::string_view, std::string_view)

(in the box)

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`

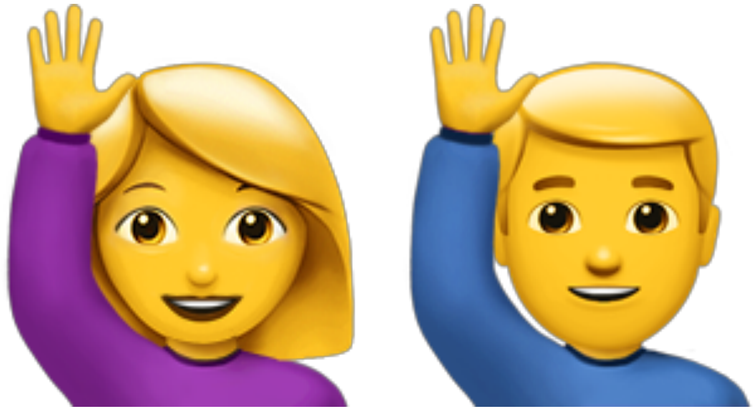
(*maybe a string literal*)

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

Let's see...

clang-tidy

Who used clang-tidy ?



+

->

LLVM
clang-tidy
clang-format

Visual Studio
2015/2017

Clang Power Tools

www.clangpowertools.com

FREE / Open source

clang-tidy

Extra Clang Tools 7 documentation

CLANG-TIDY - CLANG-TIDY CHECKS

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

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

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 ?

#godboltallthethings

GODBOLT ALL THE THINGS !!!

https://godbolt.org

COMPILER EXPLORER

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/g/CVXK9m

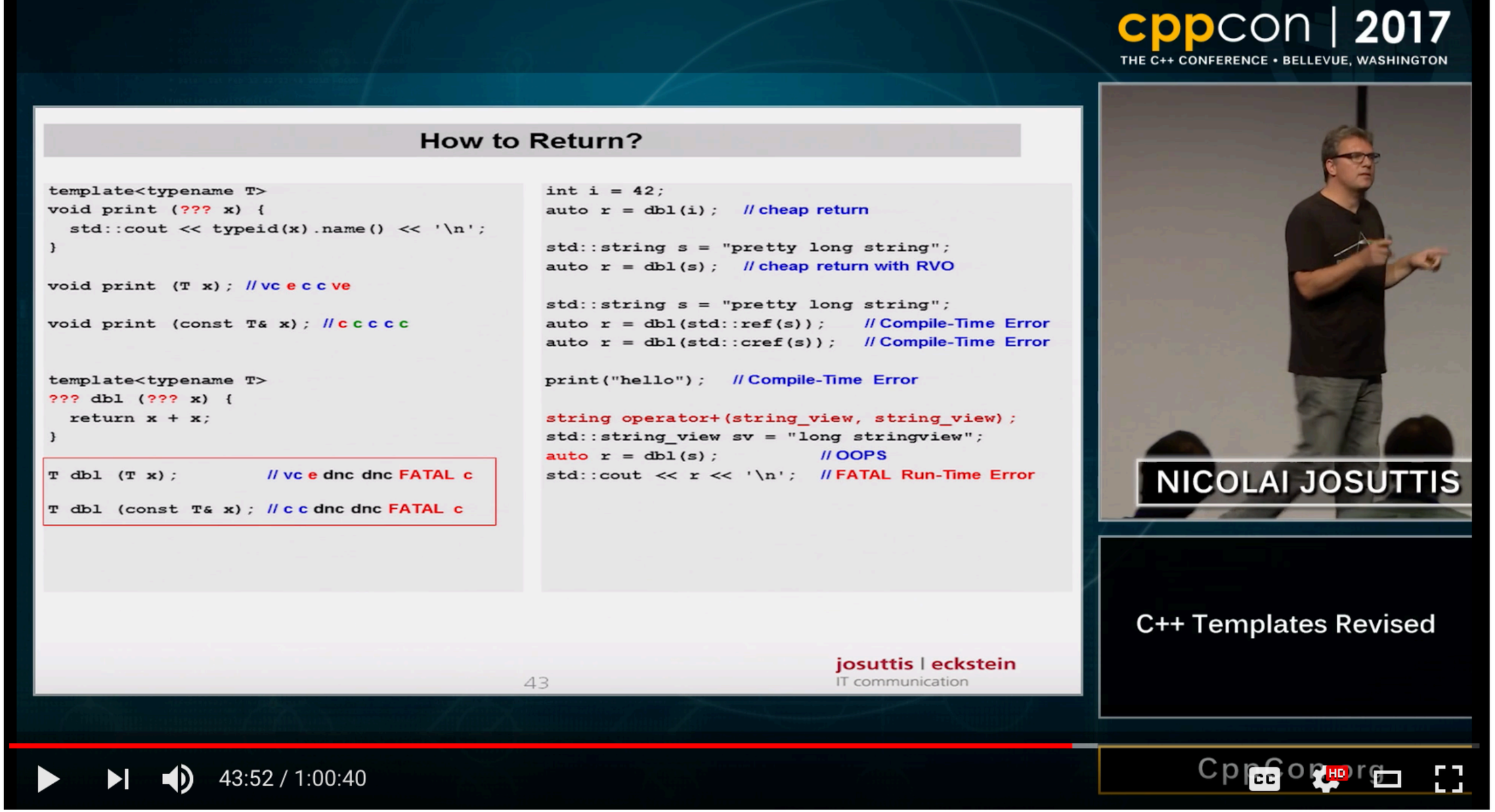
```
s1: 3 'abc'

s2: 8 'abcdef'
```

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...



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

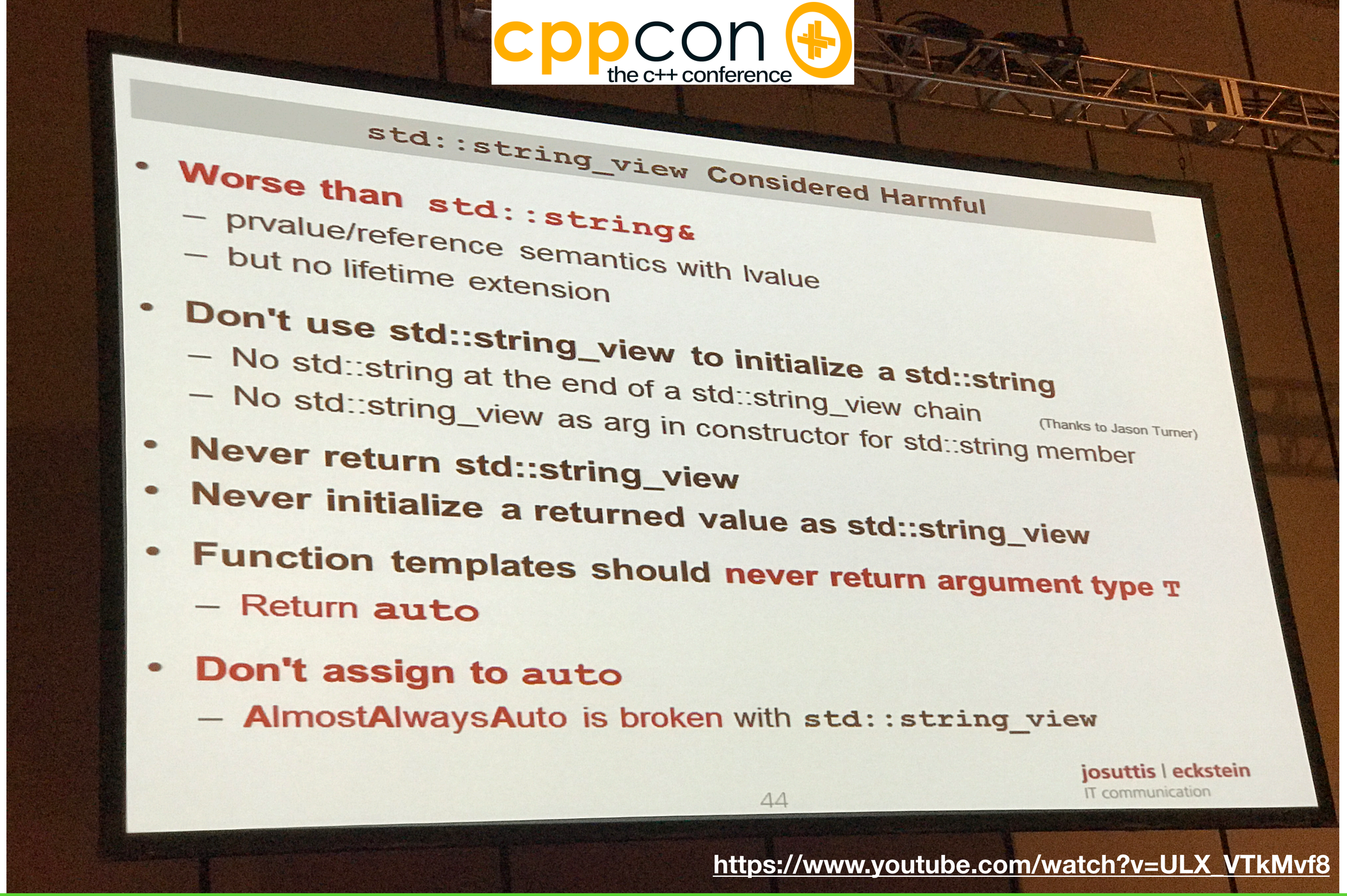
Nicolai Josuttis:

```
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`"



https://www.youtube.com/watch?v=ULX_VTkMvf8

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.

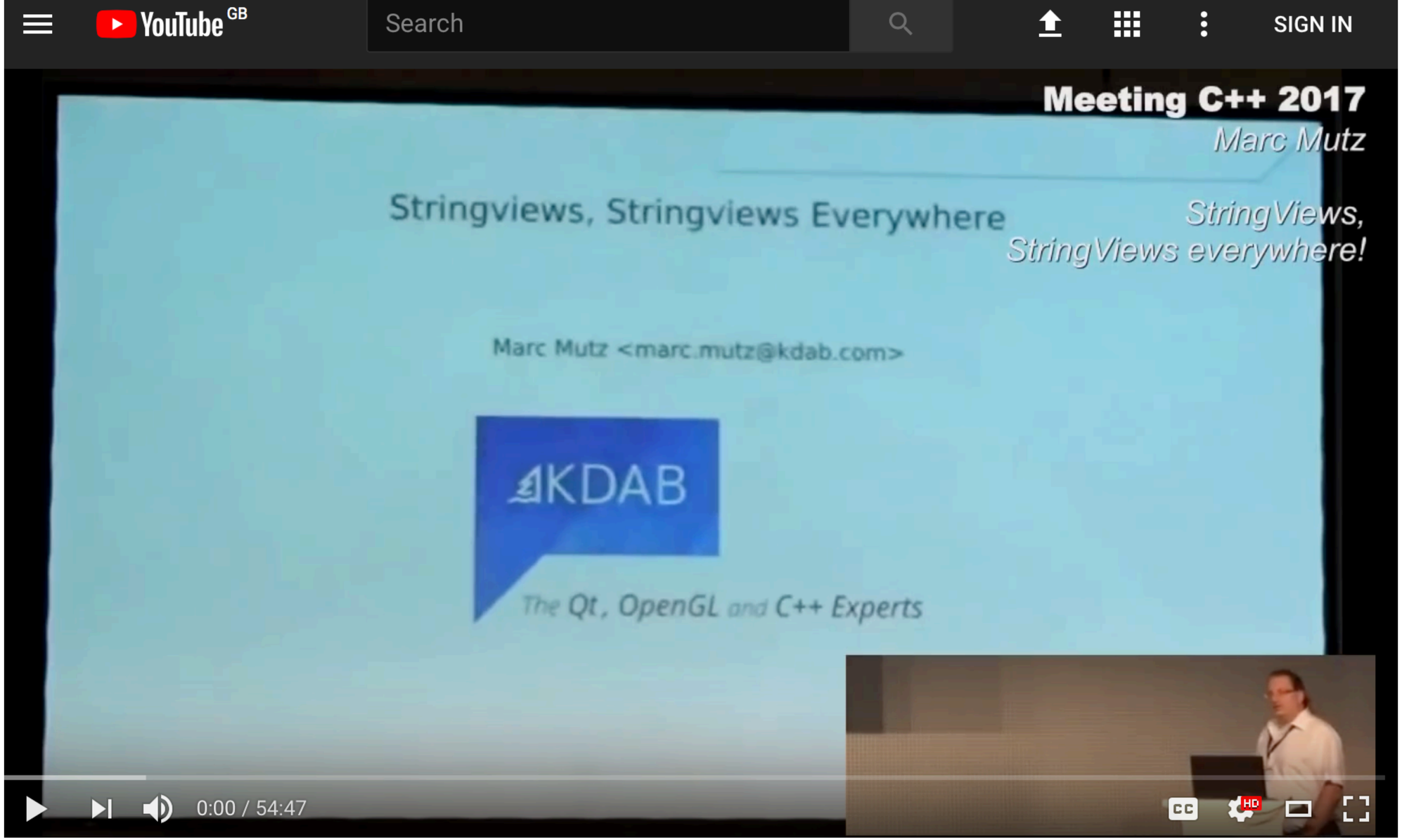
Classes not Having Value Semantics

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

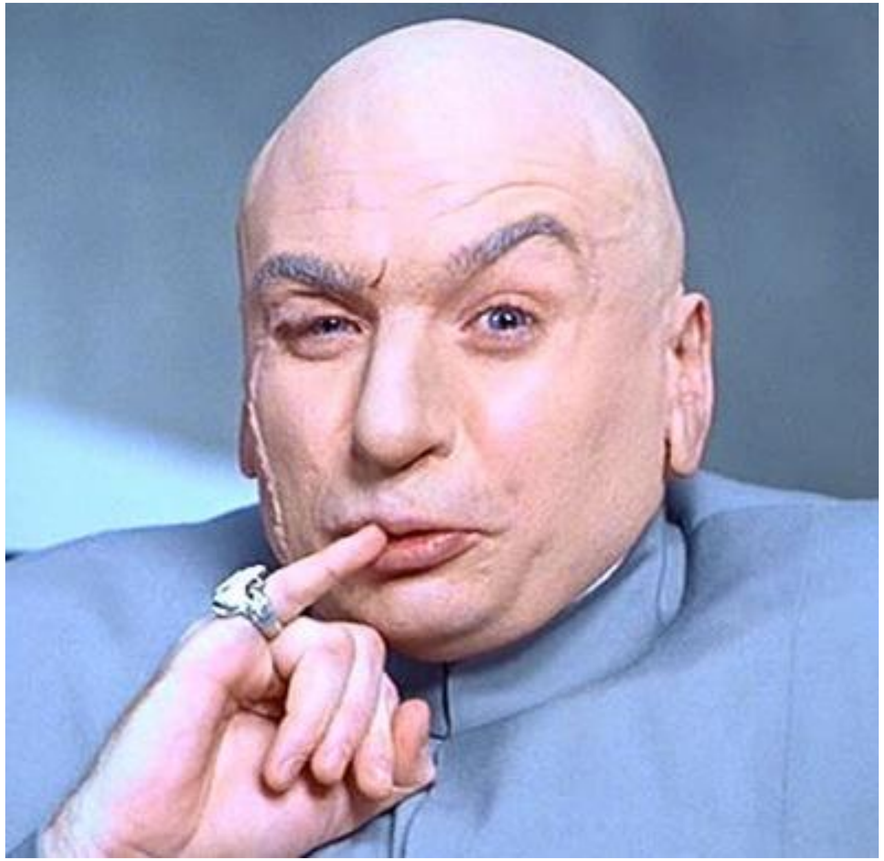
A few more months passed...



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 Krzemieński's C++ blog

String’s competing constructors

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

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/

`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/

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/

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

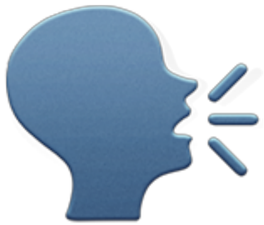
https://www.youtube.com/channel/UCsefcSZGxO9lTBqFbsV3sJg/

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

Jon Kalb @_JonKalb

Phil Nash @phil_nash

Questions



 **@ciura_victor**