# Unraveling string_view:
## Basics, Benefits, and Best Practices

JASMINE LOPEZ &
PRITHVI OKADE

Cppcon
The C++ Conference

2024
September 15 - 20

# Topics

- Motivation
- Performance benefits & basics
- string_view: Constructors, useful functions
- string vs. string_view and their interoperability
- When to use string_view
- Using string_view safely
- Intro to span
- span vs. string_view
- Case study of an optimization using string_view.

# Motivation

- Consider a function foo which operates on an immutable string.
- In C++ we generally will create it with following signature.

1

```
void foo(const std::string& str);
```

```
string existing_str;
foo(existing_str);
```

```
foo("hello this is a long string");
```

This will do memory allocation.

If this was in a performance sensitive portion of the code and we did not want memory allocation, we may need to write alternate methods.

2

```
void foo(const char* str, size_t len);
```

For code reuse “1” and “3” will end up calling “2”.

3

```
void foo(const char* str);
```

And the code will miss the niceties of using the string API set.

**string_view** helps in resolving this problem elegantly.

# Motivation

Instead of the following 3 functions:

```
void foo(const std::string& str);
void foo(const char* str, size_t len);
void foo(const char* str);
```

```
std::string s("hello");
const char* p_str = s.c_str();
const size_t p_str_size = s.size();

foo(s);
foo({p_str, p_str_size});
foo(p_str);
```

We can just write:

```
void foo(std::string_view sv);
```

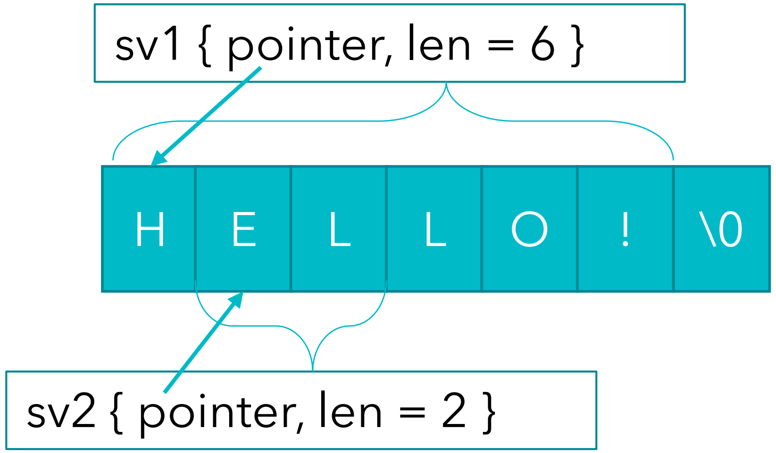
```
void foo(std::string_view sv) {
  cout << sv << '\n';
}
```

```
hello
hello
hello
```

Apart from this convenience, **string_view** also provides performance benefits which we will see shortly.

# Basics

- string_view does not allocate any memory.
- It consists of a) pointer to string and b) length.




Unlike std::string, std::string_view is NOT null terminated*.

*There are non-standard workarounds for null-terminated string_views (i.e. Chromium's cstring_view)

# Basics

```
std::string s("hello");

cout << s << ", " << hex << (void*)s.c_str()
<< ", len: " << s.size() << '\n';
```

```
hello, 0x7ffd8887d938, len: 5
```

```
void print(string_view sv) {
  cout << sv << ", " << hex
       << (void*)sv.data()
       << ", len: " << sv.size() << '\n';
}

print(s);
```

```
hello, 0x7ffd8887d938, len: 5
```

```
const char* pchar = s.c_str();
print(pchar);
```

```
hello, 0x7ffd8887d938, len: 5
```

```
print({pchar + 1, 2});
```

```
el, 0x7ffd8887d939, len: 2
```

Notice that **cout** understands **string_view** and prints it correctly. Only 2 characters.

```
string_view sv{pchar + 1, 2};
puts(sv.data());
```

Don't do this when you are using string_view

```
ello
```

# Performance benefits

- The fact that string_view does not allocate memory can be used to gain performance in some scenarios. E.g., String splitting.

```
vector<string> split_string(const string& str, char delim) {
  vector<string> splits;
  size_t index = 0;
  while (true) {
    const auto found_index = str.find(delim, index);
    if (found_index != string::npos) {
      splits.emplace_back(str.substr(index, found_index - index));
      index = found_index + 1;
    } else {
      splits.emplace_back(str.substr(index));
      break;
    }
  }
  return splits;
}
```

```
template <typename Collection>
void print(const Collection& coll) {
  for (size_t i = 0; i < coll.size(); ++i) {
    cout << coll[i];
    if (i != coll.size() - 1)
      cout << '+';
  }
  cout << '\n';
}
```

```
string s("hello|how|are|you");
print(split_string(s, '|'));
```

```
hello+how+are+you
```

There are a lot of memory allocations for strings. Each **substr** call will cause a memory allocation.

**string_view** can be a good replacement for this scenario.

# Performance benefits

- The fact that string_view does not allocate memory can be used to gain performance in some scenarios. E.g., String splitting.

```
vector<string_view> split_string_sv(string_view str, char delim) {
  vector<string_view> splits;
  size_t index = 0;
  while (true) {
    const auto found_index = str.find(delim, index);
    if (found_index != string::npos) {
      splits.emplace_back(str.substr(index, found_index - index));
      index = found_index + 1;
    } else {
      splits.emplace_back(str.substr(index));
      break;
    }
  }
  return splits;
}
```

```
template <typename Collection>
void print(const Collection& coll) {
  for (size_t i = 0; i < coll.size(); ++i) {
    cout << coll[i];
    if (i != coll.size() - 1)
      cout << '+';
  }
  cout << '\n';
}
```

```
string s("hello|how|are|you");
print(split_string_sv(s, '|'));
```

```
hello+how+are+you
```

This removes all unnecessary string allocations.

The standard library has kept the interface of string_view very similar to string. So very few modifications are needed.

# Type definition

```
namespace std {
template <class CharT, class Traits = std::char_traits<CharT>>
class basic_string_view;
}
```

It is present in header **<string_view>**

There are some common typedefs

```
namespace std {
typedef basic_string_view<char> string_view;
typedef basic_string_view<char16_t> u16string_view;
typedef basic_string_view<char32_t> u32string_view;
typedef basic_string_view<wchar_t> wstring_view;

// C++ 20
typedef basic_string_view<char8_t> u8string_view;
}  // namespace std
```

# C++17 Constructors

```
constexpr basic_string_view() noexcept;
constexpr basic_string_view(const basic_string_view& other) noexcept = default;
constexpr basic_string_view(const CharT* s, size_type count);
constexpr basic_string_view(const CharT* s);
```

No move constructor, so what happens when someone uses std::move?

```
string_view sv{"hello"};
auto s{move(sv)};
```

The move constructor is “not declared” instead of “deleted”, so it gets replaced with copy constructor.

How does the following code work?

```
string s{"hello"};
string_view sv(s);
```

`string` has a conversion operator to `string_view`.

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

# C++20 Constructors

```
template <class It, class End>
constexpr basic_string_view(It first, End last);
```

This iterator constructor was added in C++20.

```
std::vector v = {'s', 'h', 'e', 'l', 'l'};
std::string_view sv{v.begin() + 1, v.end()};
std::cout << sv << '\n';
```

```
hell
```

```
constexpr char kStr[] = "shell";
std::string_view sv{kStr + 1, kStr + 3};
std::cout << sv << '\n';
```

```
he
```

```
template <class It, class End>
constexpr basic_string_view(It first, End last);
```

The types “It” and “End” don’t need to be same.

These overloads participates in overload resolution only if:

- It satisfies contiguous_iterator
- End satisfies sized_sentinel_for for It
- std::iter_value_t<It> and CharT are the same type, and
- End is not convertible to std::size_t.

# C++23 Constructors

```
template <class R>
constexpr explicit basic_string_view(R&& r);
```

This range constructor was added in C++23.

```
std::vector v = {'h', 'e', 'l', 'l', 'o'};
std::string_view sv{v};
std::cout << sv << '\n';
```

```
hello
```

This overload participates in overload resolution only if:

- **std::remove_cvref_t<R>** is not the same type as **std::basic_string_view.**
- **R** models **contiguous_range** and **sized_range.**
- **ranges::range_value_t<R>** and **CharT** are the same type.
- **R** is not convertible to **const CharT***, and
- Let **d** be a *lvalue* of type **std::remove_cvref_t<R>**,
  **d.operator ::std::basic_string_view<CharT, Traits>()** is not a valid expression.

# C++23 Constructors

```
// C++20 code.
std::string_view sv{nullptr};
```

This code compiles in C++20 and then crashes at runtime.

```
// C++23 code.
std::string_view sv{nullptr};
```

```
error: call to deleted constructor of 'std::string_view' (aka 'basic_string_view<char>')
 std::string_view sv{nullptr};
                  ^ ~~~~~~~~~
./include/c++/v1/string_view:348:3: note: 'basic_string_view' has been explicitly marked deleted here
   basic_string_view(nullptr_t) = delete;
```

```
constexpr basic_string_view(std::nullptr_t) = delete;
```

This constructor was deleted in C++23.

```
const char* p_str = nullptr;
std::string_view sv{p_str};
```

This code compiles in C++23 and then crashes at runtime.

# operator""sv: Removing strlen

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

This constructor uses **"strlen"**. How do we remove that?

The standard defined a literal operator for this purpose.

```
constexpr std::string_view
    operator "" sv(const char* str, std::size_t len) noexcept;
```

Usage:

```
auto sv = "hello"sv;
string_view sv1 = "hello"sv;
```

This avoids `strlen` usage.

This is defined in <string_view> in following namespace:

```
namespace std {
inline namespace literals {
inline namespace string_view_literals {
// Defined here.
}
}  // namespace literals
}  // namespace std
```

Any of the following using can be used to use **sv**:

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

Why inline namespaces are used here?

Read [this answer] from Howard Hinnant.

# operator""sv: Nuance

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

cout << s1 << ", " << s1.size() << '\n';
cout << s2 << ", " << s2.size() << '\n';
```

```
abc, 3
abcdef, 8
```

Since operator""sv does not need to do strlen, it can contain embedded \0's.

# string_view vs. string

- **string** owns memory, **string_view** does not.
- **string** is always null terminated, **string_view** may not.
- `string::data()` can never return `nullptr`, `string_view::data()` can.

```
void foo(const std::string& s) {
  const auto n = strlen(s.data());
  // Do stuff with n.
}
```

```
void foo(string_view s) {
  const auto n = strlen(s.data());
  // Do stuff with n.
}
```

Don't do this.

```
void foo(string_view s) {
  if (s.empty())
    return;
  const auto n = s.size();
  // Do stuff with n.
}
```

Always check for empty before using

Always use size() to figure out the range to operate on. Never use just data().

# string_view vs. string: library functions

- string_view is "mostly" non-mutable, so it **does not** have the following "mutating" methods present in string:
  - reserve
  - shrink_to_fit
  - clear
  - insert
  - erase
  - push_back
  - pop_back
  - append
  - operator+=
  - replace
  - resize
- Mutators
  - operator=
  - swap
- Following mutators are not present in string
  - remove_prefix
  - remove_suffix

```
auto sv = "_name_"sv;
sv.remove_prefix(1);
cout << sv << '\n';
sv.remove_suffix(1);
cout << sv << '\n';
```

```
name_
name
```

# string_view vs. string: library functions

- Some other methods in string which are not present in string_view
  - c_str: Since string_view cannot provide null terminated string guarantee.
  - capacity: No need, since there is no reserve.
  - get_allocator: No need, since it does not allocate memory.

- substr method in string_view is O(1)
  - In string's it's O(N). And can allocate memory.

# string_view vs. string: operator[] & at

- operator[] and at() behaves differently in string_view
  - They are both read-only.
  - This is because string_view can point to string literals and attempt to update that is undefined behavior.

```
string s("hello");
cout << s[0] << s.at(1) << '\n';
s[0] = 'H';
s.at(1) = 'E';
```

```
auto s = "hello"sv;
```

```
cout << s[0] << s.at(1) << '\n';
```

```
s[0] = 'H';
s.at(1) = 'E';
```

```
error: cannot assign to return value because function
'operator[]' returns a const value
s[0] = 'H';
```

```
error: cannot assign to return value because function
'at' returns a const value
s.at(1) = 'E';
```

- Out of bounds access behavior in Standard Template Library (STL)
  - at - throws an exception
  - operator[] - undefined behavior

```
std::string_view sv("hello");
// Out of bound access.
std::cout << sv.at(100) << '\n';
```

```
libc++abi: terminating due to uncaught exception of type std::out_of_range:
string_view::at
Program terminated with signal: SIGSEGV
```

```
std::string_view sv("hello");
// Out of bound access.
std::cout << sv[100] << '\n';
```

```
T
```

# string_view / string: Interoperability

**string** can be automatically converted to **string_view**.

```
std::string s("hello");
std::string_view sv = s;
```

This works because **string** has a conversion operator to **string_view**

```
operator std::basic_string_view<CharT, Traits>() const noexcept;  // C++17.
constexpr operator std::basic_string_view<CharT, Traits>()
    const noexcept;  // C++20.
```

**string_view** to **string** conversion must be explicit.

```
std::string_view sv("hello");
std::string s = sv;
```

```
error: no viable conversion from 'std::string_view' (aka 'basic_string_view<char>') to
'std::string' (aka 'basic_string<char>')
    std::string s = sv;
```

```
void Foo(const std::string& s) {}
```

```
int main() {
  std::string_view sv("hello");
  Foo(sv);
}
```

# string_view / string: Interoperability

**string_view** to **string** conversion must be explicit.

```
std::string_view sv("hello");
std::string s = sv;
```

```
error: no viable conversion from 'std::string_view' (aka 'basic_string_view<char>') to
'std::string' (aka 'basic_string<char>')
    std::string s = sv;
```

```
// C++17
template <class StringViewLike>
explicit basic_string(const StringViewLike& t,
                      const Allocator& alloc = Allocator());
// From C++20
template <class StringViewLike>
constexpr explicit basic_string(const StringViewLike& t,
                                const Allocator& alloc = Allocator());
```

```
std::string_view sv("hello");
std::string s(sv);
```

```
void Foo(const std::string& s) {}
```

```
int main() {
  std::string_view sv("hello");
  Foo(sv);
}
```

```
int main() {
  std::string_view sv("hello");
  Foo(std::string{sv});
}
```

What is **StringViewLike**?

It's a type which **can be** converted to std::string_view

But the type **cannot be** converted to const char*

```
struct A {
  A(const std::string& s) : s_(s) {}
  operator std::string_view() const { return s_; }
  std::string s_;
};
```

```
A a("hello");
std::string s(a);
```

# string_view / string: Interoperability

**string_view** to **string** conversion must be explicit.

```
// C++17
template <class StringViewLike>
explicit basic_string(const StringViewLike& t,
                      const Allocator& alloc = Allocator());
// From C++20
template <class StringViewLike>
constexpr explicit basic_string(const StringViewLike& t,
                                const Allocator& alloc = Allocator());
```

```
std::string_view sv("hello");
std::string s = sv;
```

```
std::string_view sv("hello");
std::string s(sv);
```

```
void Foo(const std::string& s) {}
```

```
int main() {
  std::string_view sv("hello");
  Foo(sv);
}
```

```
int main() {
  std::string_view sv("hello");
  Foo(std::string{sv});
}
```

Why does it need to be explicit?

C++ standard folks felts that since std::string allocates memory, so, developer needs to be explicit.

Not every developer agrees with this stance. Check this stackoverflow post.

# string_view / string: Interoperability

**string_view** to **string** conversion must be explicit.

Here's a scenario which can explain the decision.

```
void Foo(const std::string& s);
void Bar(const std::string& s);
```

If implicit conversion was allowed, following would compile.

```
void Baz(std::string_view sv) {
  Foo(sv);
  Bar(sv);
}
```

This would cause 2 memory allocations, 1 for each string.

Without that it is more likely developers would write more optimal code.

```
void Baz(std::string_view sv) {
  Foo(std::string{sv});
  Bar(std::string{sv});
}
```

Realize duplication and update.

```
void Baz(std::string_view sv) {
  // Construct string once and use it twice.
  const std::string s{sv};
  Foo(s);
  Bar(s);
}
```

# Where to use?

string_view can be used as a function argument to remove the need of multiple functions dealing with strings.

```
void foo(const std::string& str);
void foo(const char* str, size_t len);
void foo(const char* str);
```

```
void foo(std::string_view sv);
```

Functions accepting const string& can be replaced with string_view to remove memory allocation.

```
void foo(const std::string& str);
```

```
void foo(std::string_view sv);
```

Ensure that string_view is not converted to string later. That will cause us to lose the optimization.

# Where to use?

- string_view can used in constexpr functions.
  - string constructors are constexpr only in C++20.
- Can be used to create compile time string constants.

```
constexpr char kHello[] = "hello";
```

Can be replaced with:

```
using namespace std::string_view_literals;
constexpr auto kHello = "Hello"sv;
```

**size()** function present in **string_view** makes it easier to work and remove need for **strlen** if conversion to **string** is needed.

```
constexpr string_view kHello("Hello");
```

This form (not using operator""sv) may take a little longer to compile, since strlen equivalent is needed.

# Where to use?

```cpp
#include <string>

constexpr char kHelloStr[] = "Hello";
constexpr std::string_view kHelloSv{"Hello"};

void Foo(const std::string& s) {}

int main() {
  Foo(kHelloStr);

  // `string_view` needs explicit conversion to `string`.
  // Which is more code to type, but more efficient at runtime since `strlen`
  // call is not necessary.
  Foo(std::string{kHelloSv});
}
```

If ***std::string*** is being used in many functions, then ***std::string_view*** makes it little more cumbersome w.r.t typing for developers. But is more performant.

```cpp
#include <string>

constexpr char kHelloStr[] = "Hello";
constexpr std::string_view kHelloSv{"Hello"};

void Foo(std::string_view s) {}

int main() {
  Foo(kHelloStr);
  Foo(kHelloSv);
}
```

Converting arguments to accept std::string_view is generally better.

# Where to use?

- In some cases, to gain performance, string_view can be returned from functions. Some scenarios:
  - If the function is returning compile time constant memory.
  - If the function is returning parts of string_view which were sent in arguments to the function.

```
string_view GetConstString(EnumValue e) {
  // Returns constant string based on enum value.
  // e.g. return "enumvalue1"sv.
}
```

```
vector<string_view> SplitString(string_view str, char delim);
```

`vector<string_view>` contains `string_view`'s which refer to sections of input `str`.

This needs careful usage. Code like following will break it.

```
string foo();
const auto splitted = SplitString(foo(), '|');
// Cannot use splitted elements at the point.
```

# Pass by value or reference?

string_view is a light-weight view of a string, it's cheap to copy.

**Pass by value!**

This is also a tip provided in absl.

Check this article for 2 more beneficial scenarios. And this follow-up one for non-optimal code generated by MSVC.

# Problems with string_view

- Since string_view does not have a null-terminator guarantee, avoid using it in functions that require/look for a null terminator and avoid using it if it will eventually be converted to a cstring.

```
std::string s("hello");

const char* pchar = s.c_str();

string_view sv{pchar + 1, 2};
puts(sv.data());
```

```
ello
```

- Since string_view does not own its memory, it needs to be used carefully to avoid use-after-free scenarios.

# Problem: Assigning strings to string_view

```
string foo();
```

```
const auto s = foo();
cout << s << '\n'; // FINE
```

```
const auto& s = foo();
auto&& sr = foo();
cout << s << sr << '\n'; // FINE
```

Lifetime of returned object is extended.

```
auto& s = foo();
```

```
error: non-const lvalue reference to type 'basic_string<...>' cannot bind to a
temporary of type 'basic_string<...>'
    auto& s = foo();
```

```
string_view s = foo();
cout << s << '\n';  // CRASH / UNDEFINED BEHAVIOR AT RUNTIME.
```

# Problem: Returning string_view from functions

```
string_view foo() {
  return "hello"sv;  // This is fine.
}
```

This is fine because "hello" is part of read-only memory.

```
string_view foo() {
  string s("hello");
  return s;  // BAD.
}
```

s will be destroyed at the end of function, hence the memory point by string_view will be dangling.

```
string_view foo(const string& s) {
  return s;
}
```

```
string s("hello");
const auto sv = foo(s);
cout << sv << '\n';  // FINE
```

```
const auto sv = foo("hello");
cout << sv << '\n';  // CRASH / UNDEFINED BEHAVIOR AT RUNTIME.
```

string s's memory is destroyed at the end of the statement.

# Problem: Returning string_view from class methods

```
class A {
  string s_;

 public:
  string_view get_s() const { return s_; }

  void set_s(string s) { s_ = move(s); }
};
```

```
A a;
const auto as = a.get_s();
cout << as << '\n';  // FINE.
```

```
A a;
const auto as = a.get_s();
a.set_s("hello");
cout << as << '\n';  // CRASH / UNDEFINED BEHAVIOR AT RUNTIME.
```

The memory that “as” points to is destroyed here.

```
A get_a();

const auto as = get_a().get_s();
cout << as << '\n'; // CRASH / UNDEFINED BEHAVIOR AT RUNTIME.
```

The memory that “as” points to is destroyed here.

# Problem: Returning string_view from templates

This is an example created by Nicolai Jossutis

```
string operator+(string_view sv1, string_view sv2) {
  return string(sv1) + string(sv2);
}
```

```
const auto s = "hello"sv + "world"sv;
cout << s << '\n';
```

```
helloworld
```

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

```
cout << dbl(1) << '\n';
cout << dbl("hello"sv) << '\n'; // BAD.
```

The return value of template **dbl** is **string_view**.

So, intermediate string created by **operator+** is converted to **string_view**, which then in *dangling*, since the string is destroyed on function exit.

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

Returning auto fixes the problem because it returns **string**.

# Problem: Storing string_view as class member variable

```
struct A {
  string_view sv;
  A(string_view sv) : sv(sv) {}
};
```

```
A a1("hello"sv);
A a2("hello");
cout << a1.sv << a2.sv << '\n';  // FINE
```

```
string foo();
```

```
A a1(string("hello"));
A a2(foo());
string h("hello");
A a3(h + " world");

cout << a1.sv << a2.sv << a3.sv << '\n';  // UNDEFINED BEHAVIOR.
```

All have dangling references.

# Problem: Catching issues with Warnings as Errors

Check out clang's dangling warning as errors.

Problem: returning string_view from functions

```
string_view foo() {
  string s("hello");
  return s;  // BAD.
}
```

s will be destroyed at the end of function, hence the memory point by string_view will be dangling.

Caught with **-Wreturn-stack-address** which shows up as a default warning in clang.

```
warning: address of stack memory associated with local variable 's' returned [-Wreturn-stack-address]
    6 |   return s;
```

```
string_view foo(const string& s) {
  return s;
}
```

```
#if defined(__clang__)
#define LIFETIME_BOUND [[clang::lifetimebound]]
#else
#define LIFETIME_BOUND
#endif

string_view foo(const string& s LIFETIME_BOUND)
{
  return s;
}
```

Check out MSVC's Lifetime Rules of the C++ Core Guidelines (-WLifetime).

```
const auto sv = foo("hello");
cout << sv << '\n';  // CRASH / UNDEFINED BEHAVIOR AT RUNTIME.
```

string s's memory is destroyed at the end of the statement.

Caught with **-Wdangling**

```
warning: temporary whose address is used as value of local variable
'sv' will be destroyed at the end of the full-expression [-Wdangling]
  15 |   const auto sv = foo("hello");
```

# span: Motivation

Consider the following functions

```
void foo(int* arr, int n) {
  for (int i = 0; i < n; ++i) {
    cout << i << ' ';
  }
  cout << '\n';
}
```

It is easy to make mistake in code. It should have been arr[i].

This is the most error prone function. Since, we must depend on caller to provide a valid "n".

```
void foo(const array<int, 5>& arr) {
  for (const auto i : arr) {
    cout << i << ' ';
  }
  cout << '\n';
}
```

**array** needs exact size specification. That reduces the usability for general cases.

```
template<typename T, size_t N>
void foo(array<T, N> arr) {
  for (const auto i : arr) {
    cout << i << ' ';
  }
  cout << '\n';
}
```

This helps with both uses:

```
foo(array{1, 2, 3, 4, 5});
foo(array{1, 2, 3});
```

```
void foo(const std::vector<int>& vec) {
  for (const auto i : vec) {
    cout << i << ' ';
  }
  cout << '\n';
}
```

vector is typesafe. But needs extra memory during construction.

```
int arr[] = {1, 2, 3, 4, 5};
foo(arr, size(arr));
foo(array{1, 2, 3, 4, 5});
foo(vector{1, 2, 3, 4, 5});
```

```
0 1 2 3 4
1 2 3 4 5
1 2 3 4 5
```

Can we have some type which can consume all these contiguous containers with a single interface?

# span: Motivation

```
void foo(span<int> s) {
  for (const auto i : s) {
    cout << i << ' ';
  }
  cout << '\n';
}
```

```
int arr[] = {1, 2, 3, 4, 5};
foo(arr);
```

```
1 2 3 4 5
```

```
array arr1{1, 2, 3, 4, 5};
array arr2{1, 2, 3};
foo(arr1);
foo(arr2);
```

```
1 2 3 4 5
1 2 3
```

```
vector v{1, 2, 3, 4, 5};
foo(v);
```

```
1 2 3 4 5
```

span provides a single representation for different types of "contiguous" sequences of elements.

It also helps to decouple the interface from the actual type of contiguous sequence (C-style array, std::array, std::vector).

It is lightweight, does not allocate memory and holds only a pointer and length.

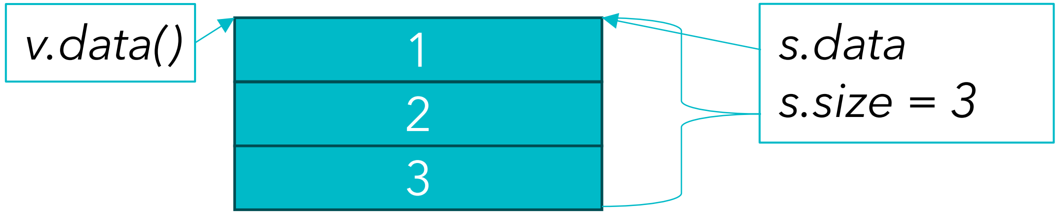
Defined in header <span> added in C++20

```
template <class T, std::size_t Extent = std::dynamic_extent>
class span;
```

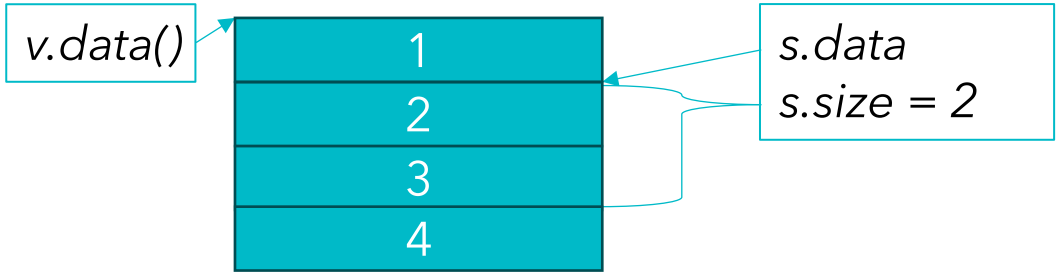
# span: Memory representation

***span*** is lightweight, does not allocate memory and holds only a pointer and length as member.

```
vector v{1, 2, 3};
// Dynamic span, initially with 3 elements.
span s{v}; // span<int>, Using deduction guide.
```




```
vector v{1, 2, 3, 4};
// Dynamic span, initially with 2 elements.
span s{v.begin() + 1, 2}; //
```




***Advantages of using a fixed span instead:***

Fixed/static ***span*** holds only a pointer, since its length is available at compile time.

# span: const and non-const

**const span:** `span<const T>`
**non-const span:** `span<T>`

Underlying container data cannot be modified with const span, whereas it can be modified with a non-const span.

```
int arr[] = {1, 2, 3};
span<const int> s_const{arr};
span<int> s_non_const{arr};
```

```
// Modifying data.
s_non_const[1] = 5;
s_non_const.front() = 6;
s_non_const.back() = 4;
sort(s_non_const.begin(), s_non_const.end());
cout << arr[0] << arr[1] << arr[2] << '\n'; // 456
```

```
// Const modifying data: Compilation error
s_const[1] = 5;
s_const.front() = 6;
s_const.back() = 4;
sort(s_const.begin(), s_const.end());
```

```
error: cannot assign to return value because function 'operator[]' returns a const value
s_const[1] = 5;

error: cannot assign to return value because function 'front' returns a const value
s_const.front() = 6;

error: cannot assign to return value because function 'back' returns a const value
s_const.back() = 4;

error: cannot assign to return value because function 'operator*' returns a const value
        *__start = _Ops::__iter_move(__child_i);
```

# span: usage scenario

span can be used as input arguments replacement for contiguous containers.

```
void foo(const int* arr, int n) {
  for (int i = 0; i < n; ++i)
    cout << arr[i] << ' ';
  cout << '\n';
}

void foo(const array<int, 3>& arr) {
  for (const auto i : arr)
    cout << i << ' ';
  cout << '\n';
}

void foo(const vector<int>& vec) {
  for (const auto i : vec)
    cout << i << ' ';
  cout << '\n';
}
```

```
const int arr[] = {1, 2, 3};
foo(arr, size(arr));
foo(array{1, 2, 3});
foo(vector{1, 2, 3});
```

```
1 2 3
1 2 3
1 2 3
```

```
void foo(span<const int> s) {
  for (const auto i : s)
    cout << i << ' ';
  cout << '\n';
}
```

span can provide “type erasure” for the individual container types.

span removes the need to always allocate memory like vector.

```
const int arr[] = {1, 2, 3};
foo(arr);
foo(array{1, 2, 3});
foo(vector{1, 2, 3});
```

```
1 2 3
1 2 3
1 2 3
```

# span: usage scenario

span can be used to pass around compile time contiguous containers.

```
// In header.
const vector<int>& GetErrorCodes();
```

```
// In source file.
const vector<int>& GetErrorCodes() {
  static const vector<int> s_error_codes{10, 30, 40};
  return s_error_codes;
}
```

```
// In header.
span<const int> GetErrorCodes();
```

```
// In source file.
span<const int> GetErrorCodes() {
  // Not magic static.
  static constexpr int kArr[]{10, 30, 30};
  return span{kArr};
}
```

Implementation can be updated to other variations, keeping the header unchanged.

```
// In source file.
namespace {
constexpr int kArr[]{10, 30, 30};
}

span<const int> GetErrorCodes() {
  return span{kArr};
}
```

```
// In source file.
namespace {
constexpr array kArr{10, 30, 30};
}

span<const int> GetErrorCodes() {
  return span{kArr};
}
```

```
// In source file.
span<const int> GetErrorCodes() {
  // Not magic static.
  static constexpr array kArr{10, 30, 30};
  return span{kArr};
}
```

# span: usage scenario

span can be used to pass around compile time contiguous containers.

Can be used with string_view to get compile time containers.

```
// In header.
const vector<string>& GetErrorStrings();
```

```
// In source file.
const vector<string>& GetErrorStrings() {
  static const vector<string> s_error_strings{
      "error 1", "error 2", "error 3"};
  return s_error_strings;
}
```

```
span<const string_view> GetErrorStrings();
```

```
// In source file.
span<const string_view> GetErrorStrings() {
  static constexpr string_view kErrors[]{"error 1", "error 2", "error 3"};
  return kErrors;
}
```

# span: unsafe usage

spans are also views which don't own memory. So, they can always lead to dangling pointer access scenarios

Returning span from function.

```
span<int> GetSpanBad() {
  vector v{1, 2, 3};
  return v;
}
```

```
const auto s = GetSpanBad();
// Cannot use elements, since they have been destroyed.
```

Using span to store other containers returned from function.

```
vector<int> GetVector() {
  return {1, 2, 3};
}
```

```
span<const int> s = GetVector();
// Cannot use elements, since they have been destroyed.
for (const auto i : s)
 cout << i << ' ';
```

```
0 0 732680208
```

Modifying a container after creating span from it.

```
vector v{1, 2, 3, 4, 5};
span s{v};
// Rellocates memory, so "s" refers to deleted memory.
v.insert(v.end(), {6, 7, 8, 9, 10});
// Bad access.
for (const auto i : s)
  cout << i << ' ';
```

```
0 0 -1365778416 21929 5
```

# span: unsafe usage

spans are views which don’t own memory. So, they can always lead to dangling pointer access scenarios

Return span from class member function.

```
class A {
 public:
  A(initializer_list<int> l) : v_(l) {}

  span<const int> GetVec() const { return v_; }
  void Add(initializer_list<int> l) { v_.insert(v_.end(), l); }

 private:
  vector<int> v_;
};
```

```
A a{1, 2, 3, 4, 5};
const auto s = a.GetVec();
// The underlying memory for span has been destroyed.
a.Add({6, 7, 8, 9, 10});
// Undefined behavior, read deleted memory.
for (const auto i : s)
  cout << i << ' ';
```

```
0 0 840699920 22077 5
```

Using span created from temporary in range based for loop.

```
for (const auto i : A{1, 2, 3, 4, 5}.GetVec()) {
  cout << i << ' ';
}
```

[Fix of Broken Range-based for loop](#)
[Compiler Support for C++23](#)

```
A a{1, 2, 3, 4, 5};
for (const auto i : a.GetVec()) {
  cout << i << ' ';
}
```

```
1 2 3 4 5
```

# span: best practices for usage

Use as argument to function which accepts any contiguous container.

```
void foo(span<const int> s);
```

Use as return value of function only when memory is backed by storage that will remain unchanged, e.g., globals.

```
span<const int> GetErrorCodes() {
  // Not magic static.
  static constexpr int kArr[]{10, 30, 30};
  return span{kArr};
}
```

Don't store as member variables of class.

Don't use it in left hand side of expression to store return values of type vector, array, etc.

```
vector<int> GetVector();
```

```
span<const int> s = GetVector();
```

Don't use span to hold non-const containers in local scope, because the container may get modified in the same scope.

```
vector v{1, 2, 3, 4, 5};
span s{v};
// Rellocates memory, so "s" refers to deleted memory.
v.insert(v.end(), {6, 7, 8, 9, 10});
```

Since it has low overhead and is cheap to copy, pass by value instead of const &.

# string_view vs. span

Both refer to contiguous sequence of elements starting at position zero with standard operations.

Both are lightweight easy-to-copy objects with a pointer and a size member.

## *string_view*

- Read-only view over strings.
- Always constant, cannot be used to modify the referred string.
- Supports string-like operations.

## *span*

- View over contiguous sequence of elements
- `span<T>` can modify contents. `span<const T>` cannot.
- More "generalized" view on containers and doesn't have string specific utilities.

string_view is the best view type for dealing with strings

# Case study for possible code changes using string_view

```cpp
// In header
const std::vector<std::string>& GetKnownHosts();
```

```cpp
// In source file.
const std::vector<std::string>& GetKnownHosts() {
  static const std::vector<std::string> known_hosts{
      "bing.com",
      "microsoft.com",
      "sharepoint.com"
  };
  return known_hosts;
}
```

```cpp
class A {
 public:
  // Other stuff.
  bool IsInMap(const std::string& host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```cpp
#include <algorithm>
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto& host)
         { return a.IsInMap(host); }
     );
}
```

# Optimization: Using span, string_view in GetKnownHosts

A better solution is to use std::span & std::string_view ☺

```
// In header
std::span<const std::string_view> GetKnownHosts();
```

```
// In source file.
std::span<const std::string_view> GetKnownHosts() {
  static constexpr std::string_view kKnownHosts[] = {
      "bing.com", "microsoft.com", "sharepoint.com"};
  return kKnownHosts;
}
```

This **const** is needed, else there are compilation errors.

```
error: no viable conversion from returned value of type 'const std::string_view[3]' (aka 'const basic_string_view<char>[3]') to
function return type 'std::span<std::string_view>' (aka 'span<basic_string_view<char>>')
   return kKnownHosts;
          ^~~~~~~~~~~
<<TRUNCATED>>
```

# Optimization: Searching through the map

Original:

```
// In header
const std::vector<std::string>& GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(const std::string& host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto& host)
         { return a.IsInMap(host); }
     );
}
```

Updated:

```
// In header file
std::span<const std::string_view> GetKnownHosts();
```

Using the above version, we immediately run into errors

```
error: no viable conversion from 'const std::string_view' to 'const
std::string' (aka 'const basic_string<char>')
  GetKnownHosts(), [&a](const auto& host) { return a.IsInMap(host); });
                                                              ^~~~
```

A fix is to do an explicit conversion to string

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto& host)
         { return a.IsInMap(std::string{host}); }
     );
}
```

But this now causes memory allocation at runtime ☹

# Optimization: Searching through the map

Original:

```
// In header
const std::vector<std::string>& GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(const std::string& host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto& host)
         { return a.IsInMap(host); }
     );
}
```

Updated:

```
// In header file
std::span<const std::string_view> GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

Does not compile ☹

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
         { return a.IsInMap(host); }
     );
}
```

# Optimization: Searching through the map

```
// In header
std::span<const std::string_view> GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
        { return a.IsInMap(host); }
     );
}
```

```
error: no matching member function for call to 'contains'
     return host_int_map_.contains(host);
            ~~~~~~~~~~~~~~^~~~~~~~
../../third_party/libc++/src/include\map(1386,30): note:
candidate function not viable: no known conversion from
'std::string_view' (aka 'basic_string_view<char>') to 'const
key_type' (aka 'const std::string') for 1st argument
   _LIBCPP_HIDE_FROM_ABI bool contains(const key_type& __k) const
{ return find(__k) != end(); }
                            ^         ~~~~~~~~~~~~~~~~~~
../../third_party/libc++/src/include\map(1388,30): note:
candidate template ignored: requirement
'__is_transparent<std::less<std::string>, std::string_view,
void>::value' was not satisfied [with _K2 = std::string_view]
   _LIBCPP_HIDE_FROM_ABI bool contains(const _K2& __k) const {
```

It cannot convert std::string_view to std::string

We cannot make the map key std::string_view because string_view does not own its memory.

# Optimization: Searching through the map

```
// In header
std::span<const std::string_view> GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
         { return a.IsInMap(host); }
     );
}
```

Do a string conversion.

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(std::string{host});
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

But it allocates memory ☹

# Optimization: Searching through the map

```
// In header
std::span<const std::string_view> GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
        { return a.IsInMap(host); }
     );
}
```

```
error: no matching member function for call to 'contains'
    return host_int_map_.contains(host);
           ~~~~~~~~~~~~~~^~~~~~~~
../../third_party/libc++/src/include\map(1386,30): note:
candidate function not viable: no known conversion from
'std::string_view' (aka 'basic_string_view<char>') to 'const
key_type' (aka 'const std::string') for 1st argument
   _LIBCPP_HIDE_FROM_ABI bool contains(const key_type& __k) const
{ return find(__k) != end(); }
                             ^         ~~~~~~~~~~~~~~~~~~
../../third_party/libc++/src/include\map(1388,30): note:
candidate template ignored: requirement
'__is_transparent<std::less<std::string>, std::string_view,
void>::value' was not satisfied [with _K2 = std::string_view]
   _LIBCPP_HIDE_FROM_ABI bool contains(const _K2& __k) const {
```

A better fix is to use transparent comparator

# Optimization: Searching through the map with std::less

```
// In header
std::span<const std::string_view> GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
         { return a.IsInMap(host); }
     );
}
```

A better fix is to use transparent comparator

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int, std::less<>> host_int_map_;
};
```

# Optimization: Searching through the map with std::less

```
// In header
std::span<const std::string_view> GetKnownHosts();
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }

 private:
  std::map<std::string, int,
                          std::less<>> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
         { return a.IsInMap(host); }
     );
}
```

A better fix is to use transparent comparator

The std::less<> allows all the following to work:

```
auto has = false;
has = host_int_map_.contains("hello");  1
const char* p = "hello";
has = host_int_map_.contains(p);  2
const std::string_view sv("hello");
has = host_int_map_.contains(sv);
const std::string s("hello");
has = host_int_map_.contains(s);
```

**Loosely,** it works for parameter type T, where the following compile (with const / reference qualifiers).

```
bool operator<(string, T);
bool operator<(T, string);
```

1 and 2 would have worked *without* **std::less<>** too but would have allocated memory.

# Optimized solution

```
// In header
const std::vector<std::string>& GetKnownHosts();
```

```
// In header file
std::span<const std::string_view> GetKnownHosts();
```

```
// In source file.
const std::vector<std::string>& GetKnownHosts() {
  static const std::vector<std::string> known_hosts{
      "bing.com",
      "microsoft.com",
      "sharepoint.com"
  };
  return known_hosts;
}
```

```
// In source file.
std::span<const std::string_view> GetKnownHosts() {
  static constexpr std::string_view kKnownHosts[] = {
      "bing.com", "microsoft.com", "sharepoint.com"};
  return kKnownHosts;
}
```

No run time memory allocation for string, vector.

```
class A {
 public:
  // Other stuff.
  bool IsInMap(const std::string& host) const {
    return host_int_map_.contains(host);
  }
 private:
  std::map<std::string, int> host_int_map_;
};
```

```
class A {
 public:
  // Other stuff.
  bool IsInMap(std::string_view host) const {
    return host_int_map_.contains(host);
  }
 private:
  std::map<std::string, int, std::less<>> host_int_map_;
};
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto& host)
         { return a.IsInMap(host); }
     );
}
```

```
bool HasKnownHost(const A& a) {
  return std::ranges::any_of(
    GetKnownHosts(),
     [&a](const auto host)
         { return a.IsInMap(host); }
     );
}
```

# Key takeways

If you have a **const std::string** convert that to **std::string_view**.

```
const std::string c_str{"hello"};
```

→

```
static constexpr std::string_view kStr{"hello"};
```

If you have a *non-constructor* function that accepts **const std::string&** consider whether you can convert that to **std::string_view**.

```
void Foo(const std::string& str);
```

→

```
void Foo(std::string_view sv);
```

With **std::string_view** if you intend to convert to **const char***, then please:

- Keep in mind that **data()** for string_view does **not** behave the same way that **c_str()** does for string. Always assume that it does not end with **'\0'**.
- Check whether its **empty()** before using it since **data()** can return **null**.
- Use **size()**.

# Links to some articles & videos on string_view

- Articles:
  - C++17 - Avoid Copying with std::string_view
  - string_view odi et amo
  - Performance of std::string_view vs std::string from C++17
  - Speeding Up string_view String Split Implementation - Follow up from above which explains why we need to be careful when measuring performance
  - std::string_view is a borrow type - Authur O'Dwyer
  - Three reasons to pass std::string_view by value - Authur O'Dwyer
  - A footnote on "Three reasons to pass std::string_view by value" - Authur O'Dwyer
  - std::string_view: The Duct Tape of String Types - Billy O'Neal
  - Stackoverflow post on no implicit conversion from std::string to std::string_view.
  - Clang's lifetimebound Attribute
  - C++ proposal for lifetimebound
- Videos:
  - CppCon 2015: Marshall Clow "string_view"
  - StringViews, StringViews everywhere!

# Links to some references on span

- Proposal paper for span.
- std::span in C++20: Bounds-Safe Views for Sequences of Objects.
- What is a span and when should I use one?
- std::span constructors.
- C++20's Conditionally Explicit Constructors.
- std::span, the missing constructor (article).
- std::span and the missing constructor (proposal)
- Differences between std::string_view and std::span.
- Fix of Broken Range-based for loop
- Compiler Support for C++23

# Special thanks

**Victor Ciura**

Enough string_view to hang ourselves

A Short Life span<> For a Regular Mess - std::span

**Chandranath Bhattacharyya**

# Thank you! Questions?