# P3655R4: cstring_view

# Requested by LEWG

- Benchmarks comparing cstring_view to std::string_view plus std::string copy
- Comparison table for
  - Constructors
  - Embedded NULs

# Principles behind P3655

1) It is a non-allocating type that provides a type-system propagating annotation of having a null terminator at the end.
2) Type carries the at-creation checked existence of a null terminator at size() until its use. If it cannot guarantee it for some operation, it should fail to compile at the step where the guarantee is potentially lost.
3) Type works identical to `string_view` and `string` where possible, similar where feasible and fail to compile if neither of those can be done.
4) Principle of least surprise: Type fits between string and string_view. Calling functions that preserve null terminator return cstring_view, functions that do not return string_view. Converting from `string` to `string_view` gives the same result as conversion from `string` to `cstring_view` and on to `string_view`.
5) Type works in the way clients of the type will expect, in that size() and strlen(c_str()) are guaranteed to give the same result.

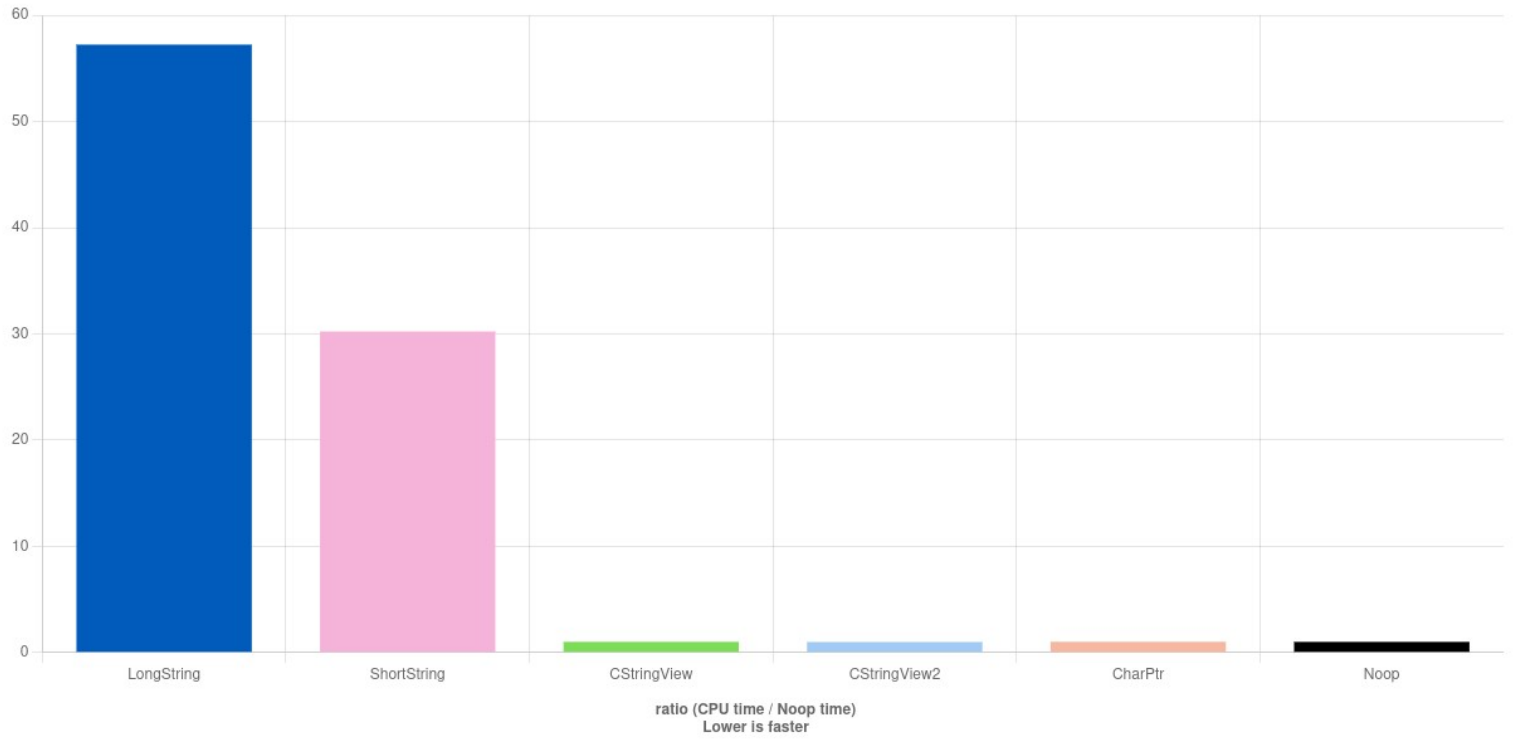
# Benchmark code

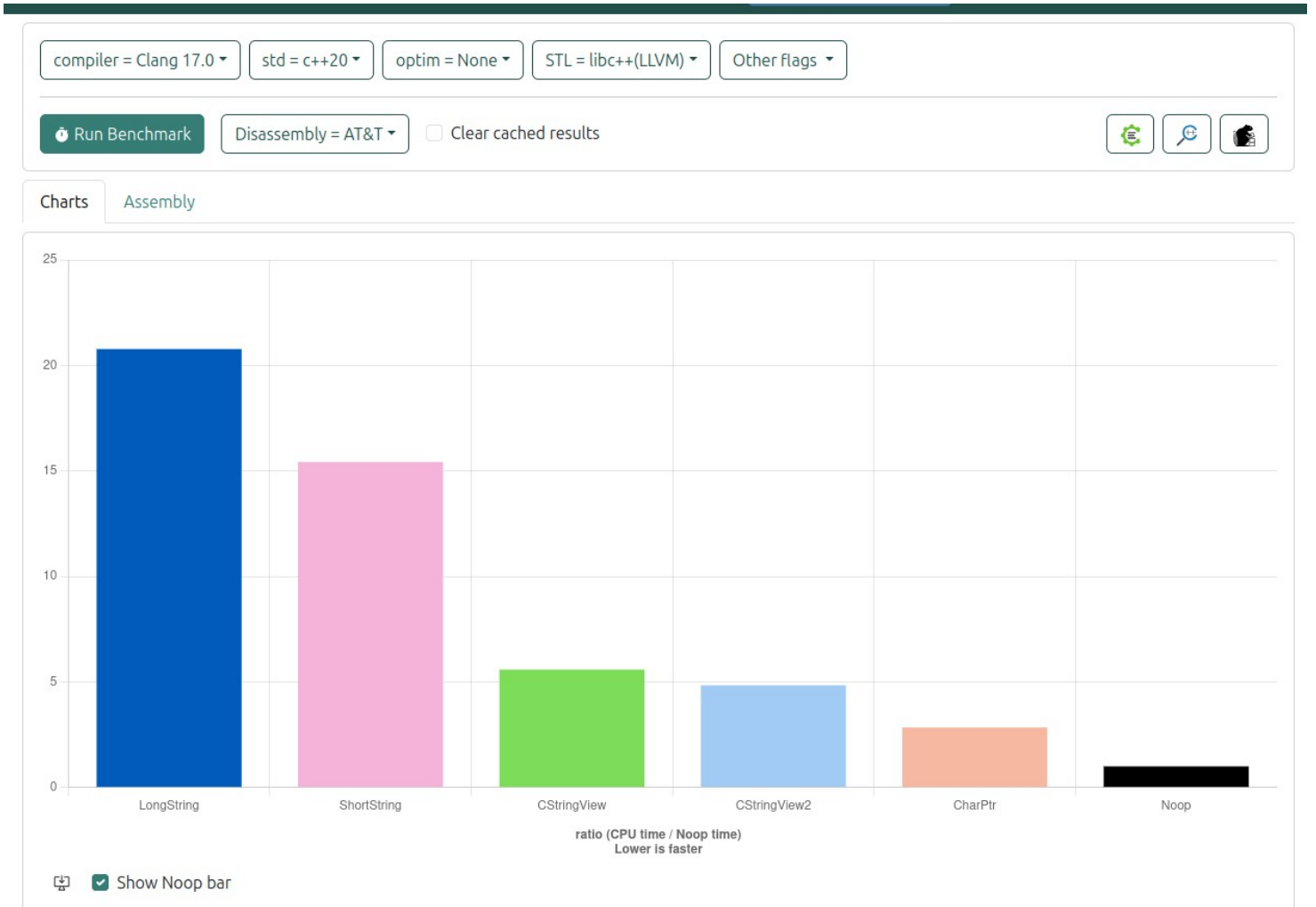
```
[[noinline]] int cfunc(const char* str) {
  return strlen(str);
}


template <typename T, typename U>
int call_c_func(T t){
  U u(t);
  return cfunc(u.c_str());
}


int l = call_c_func<std::string_view, std::string>("hello from...buffer");
int l = call_c_func<std::string_view, std::string>("hello");
int l = call_c_func<cstring_view, cstring_view>("hello from...buffer");
int l = call_c_func<cstring_view, cstring_view>("hello");

"hello from a very long string that will surely be too long to fit in the internal buffer"
```

compiler = Clang 17.0 | std = c++20 | optim = O3 | STL = libc++(LLVM) | Other flags

Run Benchmark | Disassembly = AT&T | Clear cached results

Charts | Assembly




compiler = Clang 17.0
std = c++20
optim = None
STL = libc++(LLVM)
Other flags
Run Benchmark
Disassembly = AT&T
Clear cached results
Charts
Assembly
25
20
15
10
5
0
LongString
ShortString
CStringView
CStringView2
CharPtr
Noop
ratio (CPU time / Noop time)
Lower is faster
Show Noop bar

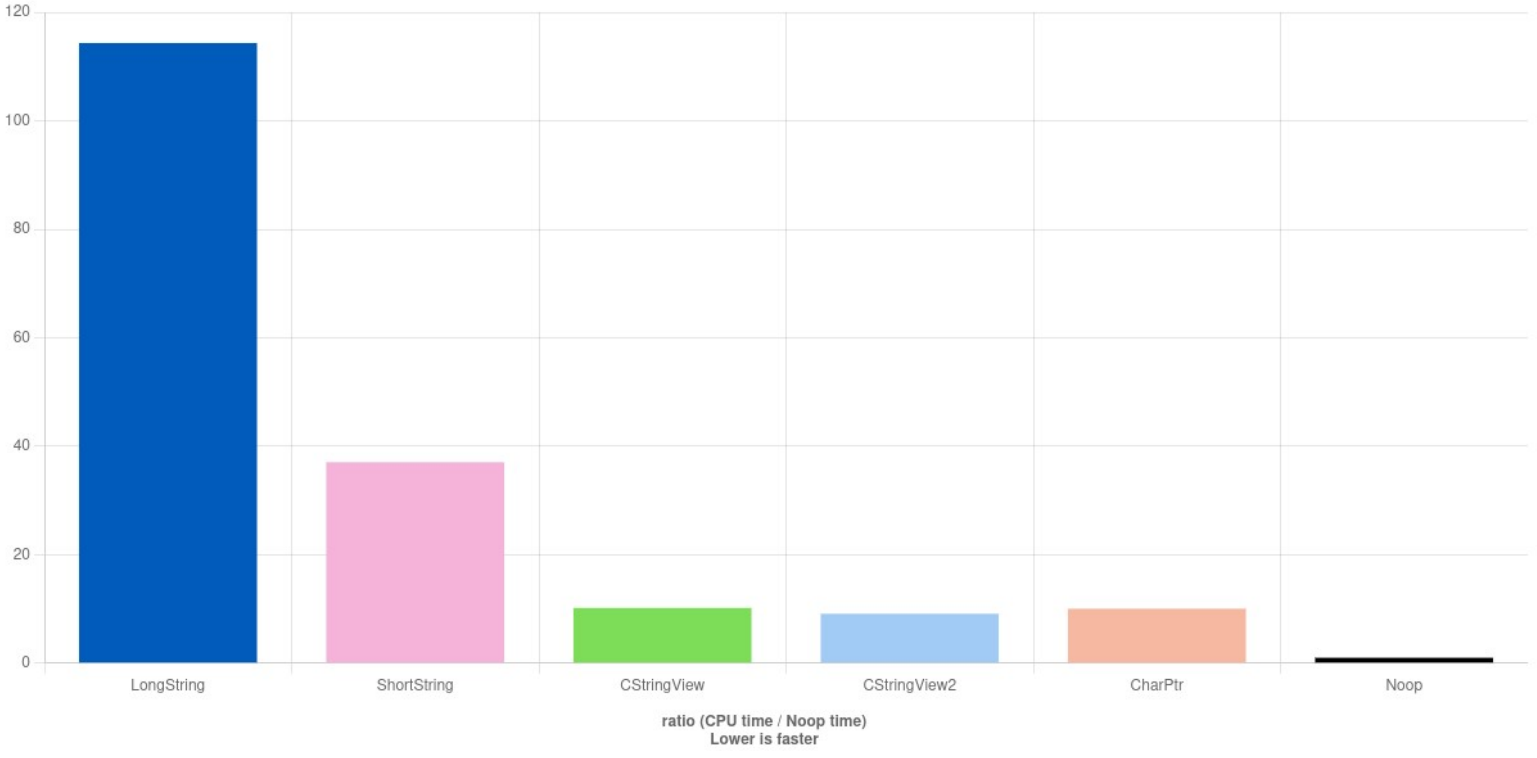
# Benchmark code

```
[[noinline]] int cfunc(const char* str) {
  benchmark::DoNotOptimize(str);
  return strlen(str);
}

template <typename T, typename U>
int call_c_func(T t){
  U u(t);
  return cfunc(u.c_str());
}


int l = call_c_func<std::string_view, std::string>("hello from...buffer");
int l = call_c_func<std::string_view, std::string>("hello");
int l = call_c_func<cstring_view, cstring_view>("hello from...buffer");
int l = call_c_func<cstring_view, cstring_view>("hello");

"hello from a very long string that will surely be too long to fit in the internal buffer"
```

compiler = Clang 17.0 | std = c++20 | optim = O3 | STL = libstdc++(GNU) | Other flags

Run Benchmark | Disassembly = AT&T | Clear cached results

Charts | Assembly

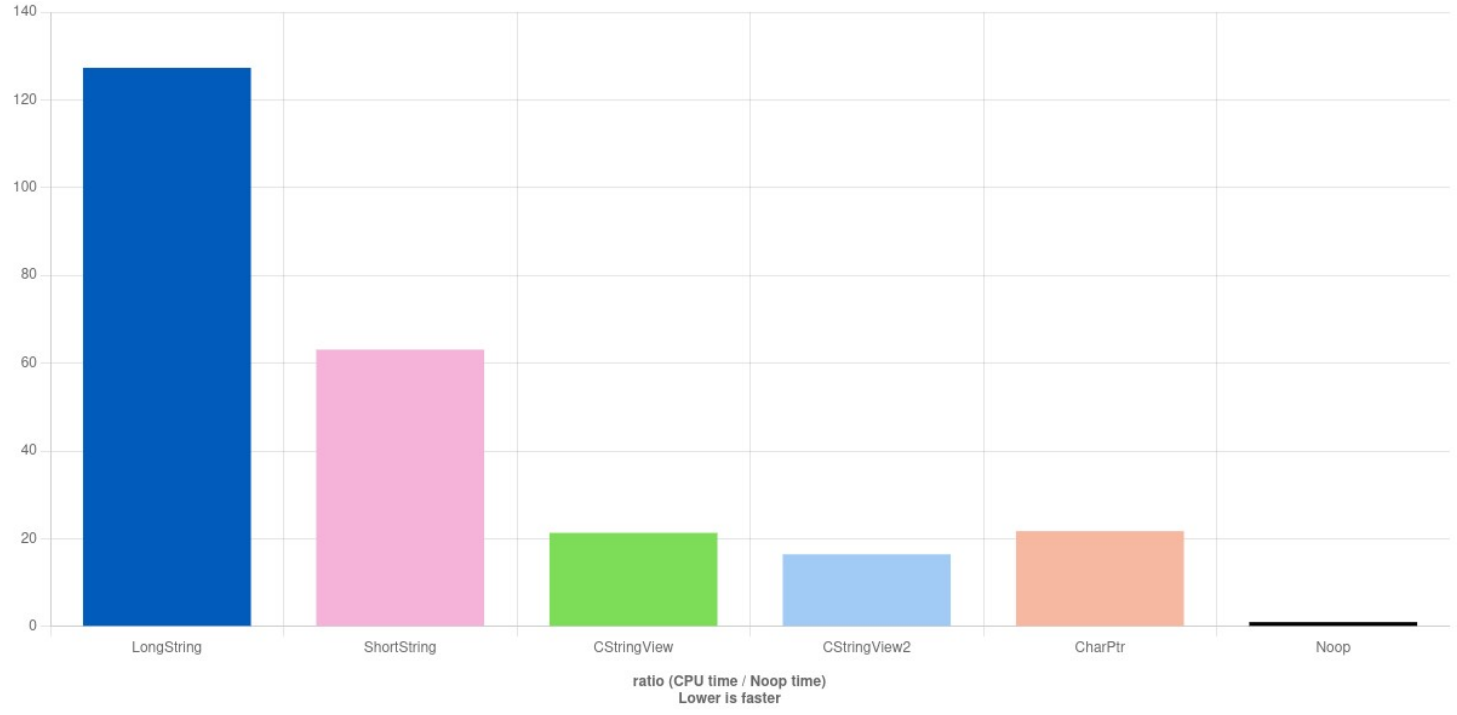



Show Noop bar

compiler = GCC 13.2 | std = c++20 | optim = O3 | STL = libstdc++(GNU) | Other flags

Run Benchmark | Disassembly = AT&T | Clear cached results

Charts | Assembly




Show Noop bar

# Benchmark

- Overhead definitely present
- Depending on API called, the overhead may be irrelevant
- For longer strings avoids
  - having allocations in the first place (MISRA)
  - nondeterministic allocation overhead
  - heap fragmentation

# Constructors

- Default
- Const char*
  - Const char (&)[N]
- Const char*, size
- Iterator pair
- Range

# (0) Default

- Default constructor refers to static single byte with 0 value
- Equivalent to std::string() in having a valid pointer and a null terminator that may not be written

# (1) const char*

- Fetches length of string with strlen()
- Identical behavior to std::string, std::string_view

- P3566 proposes making this a legacy constructor or tagged constructor wrt safety
- P3655 retains this constructor; strongly preferring keeping these issues as separate papers fixing all string types

# (1a) const char (&)[N]

- Alternate constructor that can replace `const char*` in most cases
- Must still search for first null terminator for `const char*` compatibility
- Can use strnlen to prevent avoidable buffer overruns for unterminated strings
- Can error on embedded NULs in string literals or input buffers as opposed to char* constructor
- P3655 does not propose this constructor; preferring keeping these issues as separate papers fixing all string types

# (2) const char*, size_t

- Constructs string with known size
- Identical user behavior to std::string, std::string_view
  - Does dereference at ptr[size] to verify the null terminator
  - Can construct a string with embedded nulls

# (3) Iterator pair

- `template<class It, class End>`
  `constexpr basic_cstring_view(It begin, End end);`
- Must dereference `*(begin + (end – begin))` to verify the null terminator
- Sized sentinel end pointers are supported
- Only contiguous storage
- Typical use would be `const char*` pair
- Can construct a string with embedded nulls

# (4) Range constructor

- `template<class R> constexpr basic_cstring_view(cstring_like R&& r);`
- Concept constraints to only types with `c_str()`
  - Happy to change to different concept wording that accomplishes the same goal
- Works for user types too
- Can construct a string with embedded nulls

# Embedded nulls

- cstring_view is a null *terminated* type.
- Embedded nulls have been used both for evil and for good:
  - “10.0.0.1\0.microsoft.com” to circumvent checks
  - “abc\0def\0ghi\0\0” to pass an array of strings
- Logically we know something is wrong when we get a call to c_str() with an embedded null

# Embedded nulls

- string has the same issue with c_str()
- string_view doesn’t have a c_str()
- For the benign cases data()/size() are the right interface
- Does it increase consensus to ‘fix’ this in cstring_view early?
  - Either by forcing the constructors to check
  - Or by making c_str() itself check

# “10.0.0.1\0.com”

| Constructor --> | Char*, char(&)[N] | (char*, size), iterator pair, range |
|---|---|---|
| char* | 10.0.0.1 | 10.0.0.1 |
| string | 10.0.0.1 (copy) | 10.0.0.1\0.com (copy) |
| string_view | 10.0.0.1 | 10.0.0.1\0.com |
| cstring_view (1) | 10.0.0.1 | Pre fail |
| cstring_view (2) | 10.0.0.1 | 10.0.0.1\0.com |
| cstring_view (3) | 10.0.0.1 | 10.0.0.1\0.com |

# Pass to C function

| Constructor --> | Char*, char(&)[N] | (char*, size), iterator pair, range |
|---|---|---|
| char* | 10.0.0.1 | (N/A) |
| string | 10.0.0.1 (copied) | 10.0.0.1\0.com (copied) |
| string_view | 10.0.0.1 (must copy) | 10.0.0.1\0.com (must copy) |
| cstring_view (1) | 10.0.0.1 | (Failed to construct) |
| cstring_view (2) | 10.0.0.1 | c_str() precondition failure |
| cstring_view (3) | 10.0.0.1 | 10.0.0.1\0.com |

# Downsides of each approach

| | Downsides |
|---|---|
| char* | Zero type system guarantees, ownership etc. |
| string | Always copies on create, size() != strlen(c_str()) |
| string_view | Must copy for c_str(), size() != strlen(c_str()) |
| cstring_view (1) | O(n) constructor, breaks string compat |
| cstring_view (2) | O(n) c_str(), different from string |
| cstring_view (3) | size() != strlen(c_str()) |

# Poll 1

- Desire has been expressed from SG23 to have an array constructor
- Paper authors plan to write separate paper to add array constructor to all string types
- Does adding the array constructor early in cstring_view increase consensus on it?

# Poll 2

- NUL present inside string is fundamentally incompatible with c_str()
  1) Model as a constructor postcondition?
  2) Model as c_str() precondition?
  3) Not in this paper - write separate paper for this discussion in string and cstring_view simultaneously

# Poll 3

- With changes from poll 1 and 2, forward to LWG for C++29?
  - LWG has not yet reviewed wording b/c of C++26 priority
  - LWG has indicated existing string wording is not great for basing this on
  - I have one or two LWG members happy to work with me on wording after this week

# “abc\0def\0ghi\0\0”

- Disallow at ctor
- Flags early making debugging easier
- Different from other string types
- `string→cstring_view→string_view` can fail at runtime, while `string→string_view` works
- Precondition failure from cstring_view ctor
- O(n) ctors

- Disallow at c_str()
- Flags on use, not on create
- Keeps compatibility with existing strings
- `string→cstring_view→string_view` does the same as `string→string_view`
- `“abc\0def\0ghi\0\0” from data()/size()`
- Precondition failure from c_str()
- O(n) c_str()

- Allow
- Keeps compatibility with existing strings
- `string→cstring_view→string_view` does the same as `string→string_view`
- `“abc\0def\0ghi\0\0” from data()/size()`
- Runtime different behavior
- Will bring a paper to fix all c_str() in Brno
- O(1) ctor and c_str()