STRINGS

String manipulation as a motivating example for generic programming.

K. Vladimirov, Syntacore, 2025
mail-to: konstantin.vladimirov@gmail.com

Hello, world!

- The C++ language is currently undergoing some active changes.

```
import std;

int main() {
  std::println("{0}, {1}!", "Hello", "world");
}
```

- Officially, this program has been valid since 2023.
- In reality, it barely started working in 2025 with some effort.
- We will be learning C++23, occasionally touching on C++26.

https://godbolt.org/z/11a83snWW

Hello, simplified world!

- We can use print the old way through a header.

```
#include <print>

int main() {
  std::println("{0}, {1}!", "Hello", "world");
}
```

- Or we can use a very classic C++98.

```
#include <iostream>

int main() {
  std::cout << "Hello, world!" << std::endl;
}
```

Hello, really old-school world!

- If we don't have std::print, we can emulate it via std::format.

```
#include <format>

int main() {
  std::cout << std::format("{}, {}!\n", "Hello", "world");
}
```

- And of course, we also have access to the old APIs inherited from the C language.

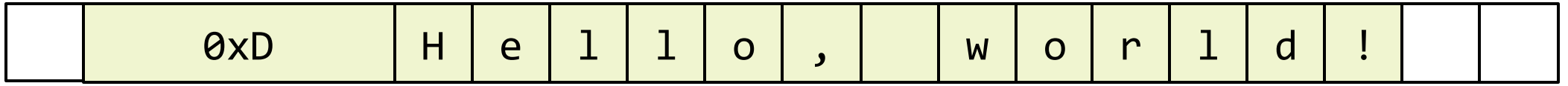
```
#include <cstdio>

int main() {
  std::printf("%s\n", "Hello, world!");
}
```


https://godbolt.org/z/9a8ax9hq4

What is a string?

```
foo("Hello, world!");
```

- How to encode the literal "Hello, world!" to use it in a program?
- Let's say we agreed on a length-prefixed string.
- What could the argument type for foo look like?



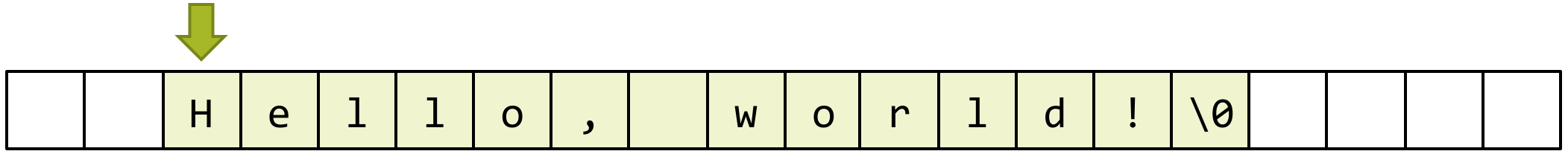
https://godbolt.org/z/rfgbsKscb

What is a string?

```
foo("Hello, world!"); // → foo(const char *);
```

- It's much easier to think about strings if they are null-terminated.
- In this case, it's just a pointer to the first element.
- `auto t = "Hello, world!"; // → const char *`

```
std::cout << sizeof("Hello, world!") << std::endl; // → 14
std::cout << sizeof(t) << std::endl; // → 8
```



https://godbolt.org/z/rfgbsKscb

Working with C-strings: <cstring> header

- strlen
- strcpy, strcat
- strcmp
- strchr, strstr
- strspn, strcspn
- strtok
- strpbrk
- strerror

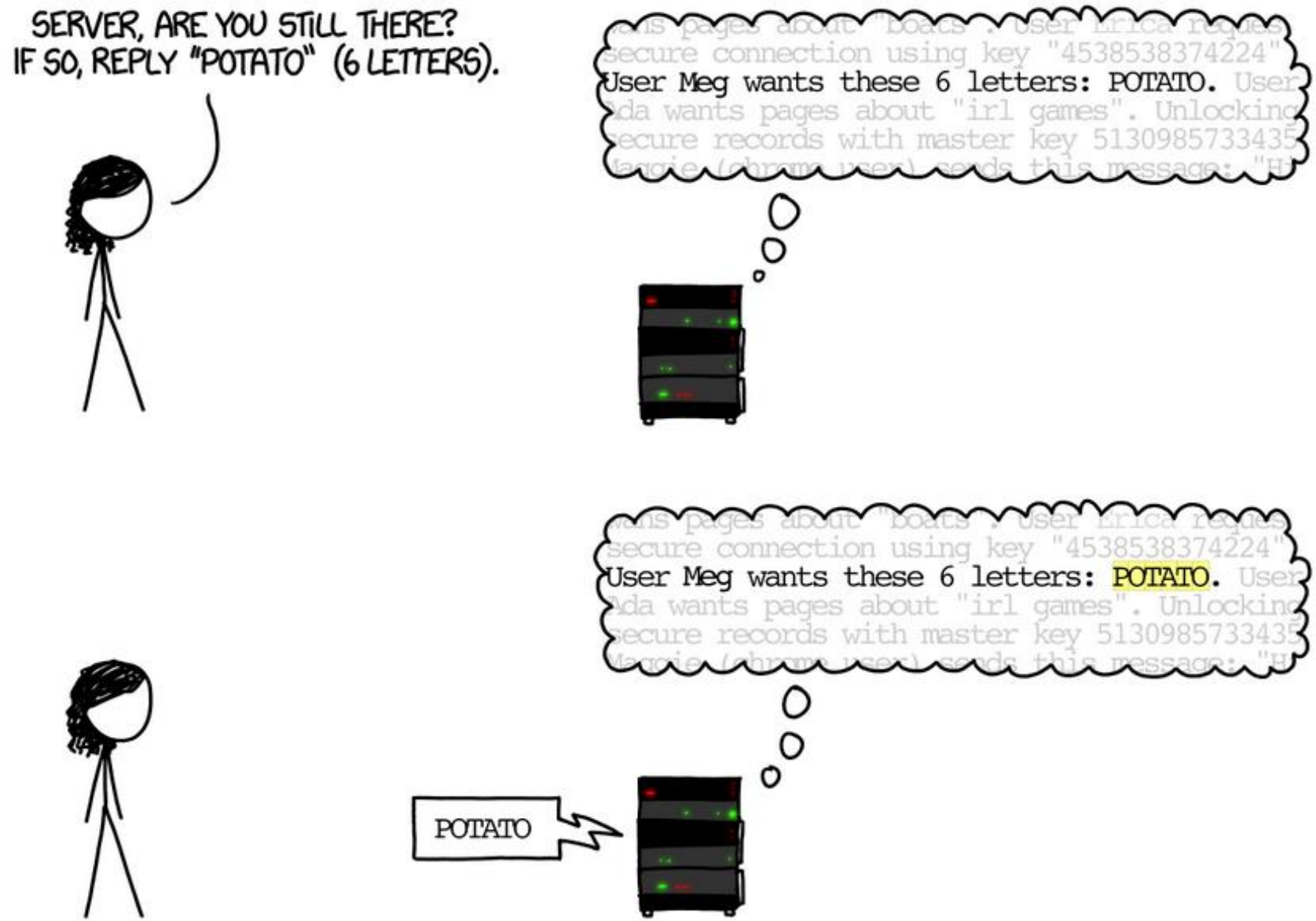
```
#include <cstring>
#include <cassert>

char astr[] = "hello";
char bstr[15];
int alen = std::strlen(astr);
assert(alen == 5);
std::strcpy(bstr, astr);
std::strcat(bstr, ", world!");
int res = std::strcmp(astr, bstr);
assert(res < 0);
```

Discussion

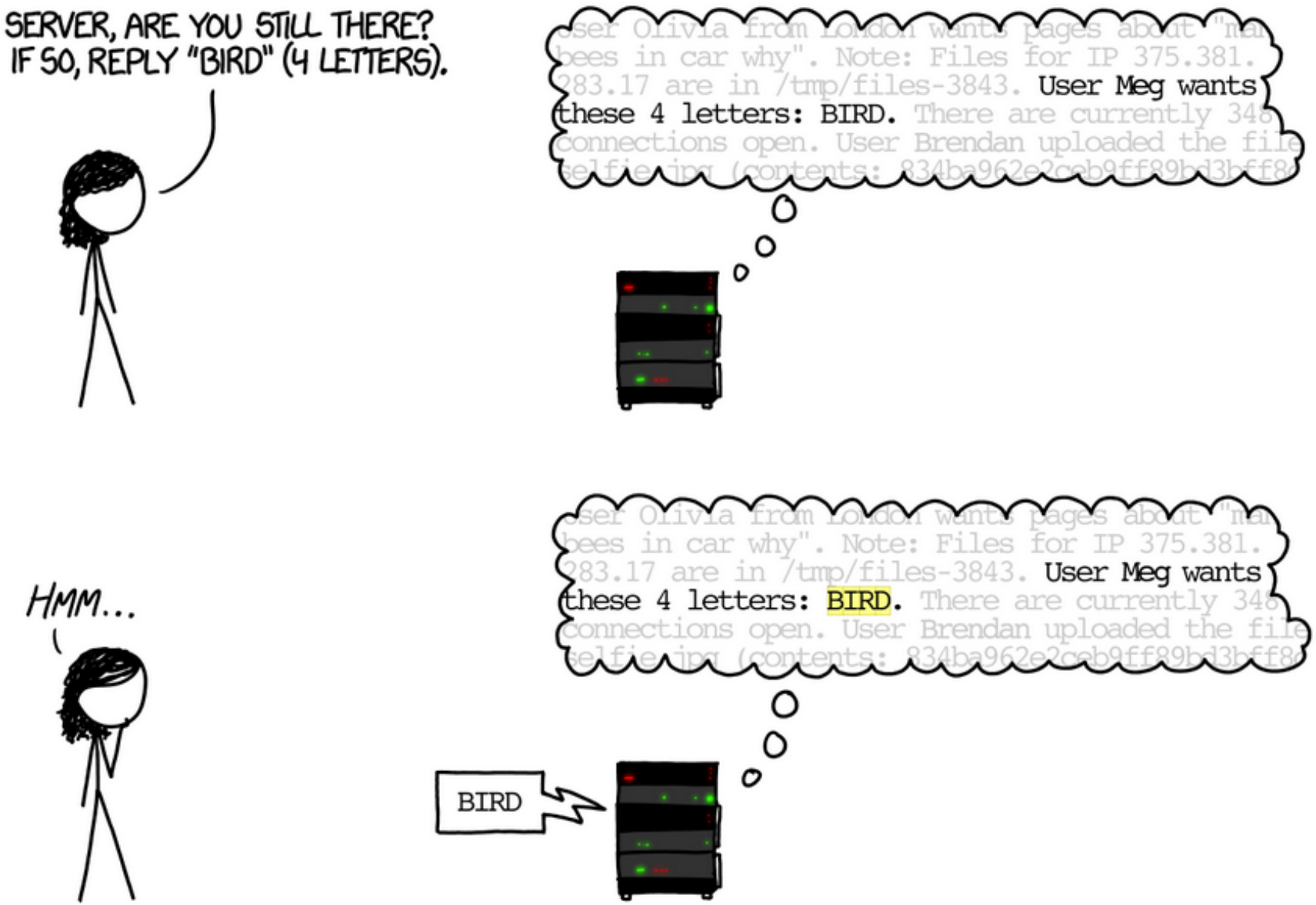
- When passing a pointer to incorrect data to a function like strcpy, we will get in return all the data up to the nearest null character.
- A solution in the C style: functions with a character limit.

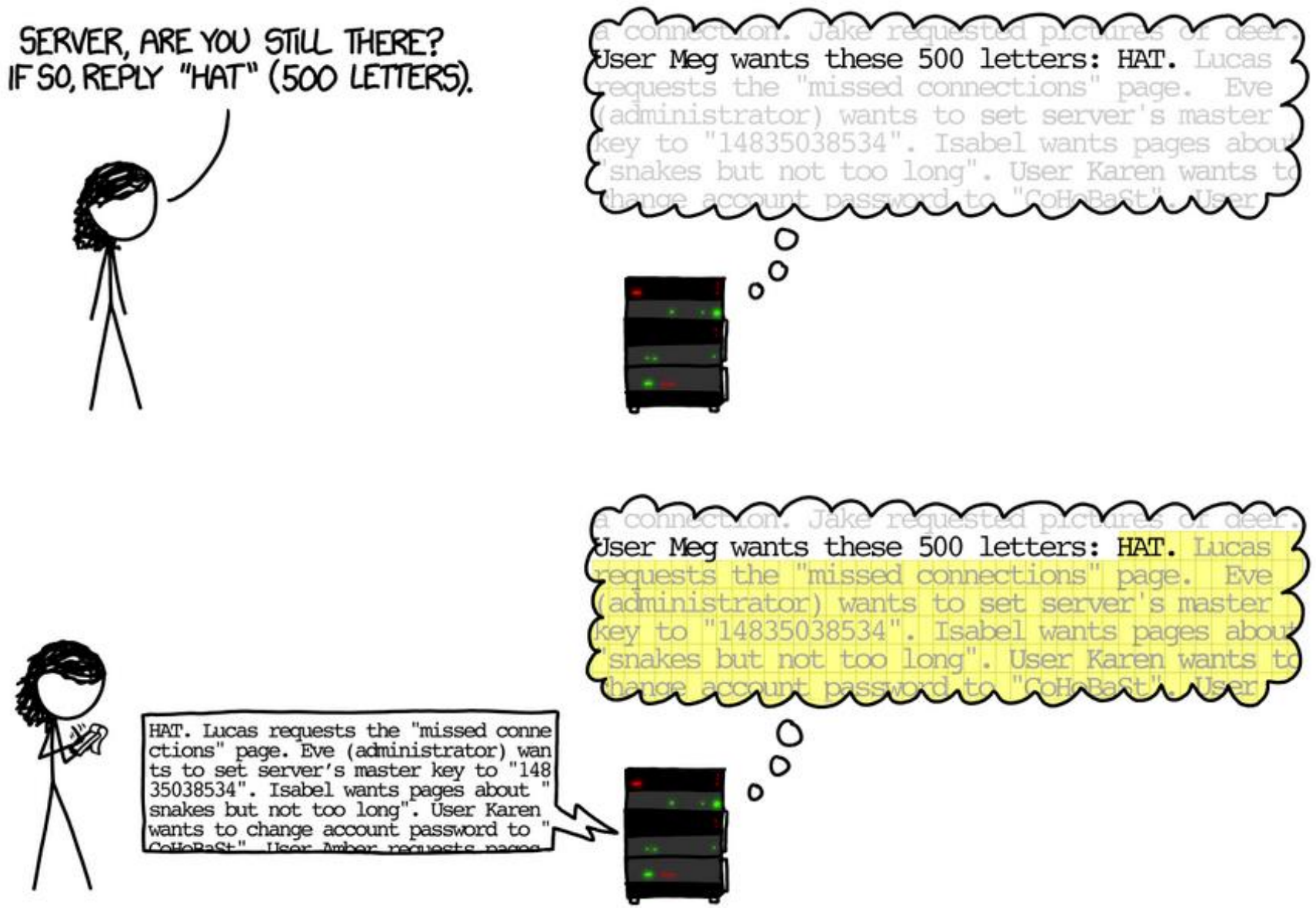
```
char* strncpy(char *dst, const char *src, size_t n);

char* strncat(char *dst, const char *src, size_t n);

int strncmp(const char *s1, const char *s2, size_t n);
```

- Does this option work?



https://xkcd.com/1354


SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 348
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2cab9ff89bd3bff8
HMM...
User Olivia from London wants pages about "ma
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 348
connections open. User Brendan uploaded the file
selfie.jpg (contents: 834ba962e2cab9ff89bd3bff8
BIRD



https://en.wikipedia.org/wiki/Heartbleed

Discussion

- The real reason for the problems is that for C strings, the length is not an **invariant**.

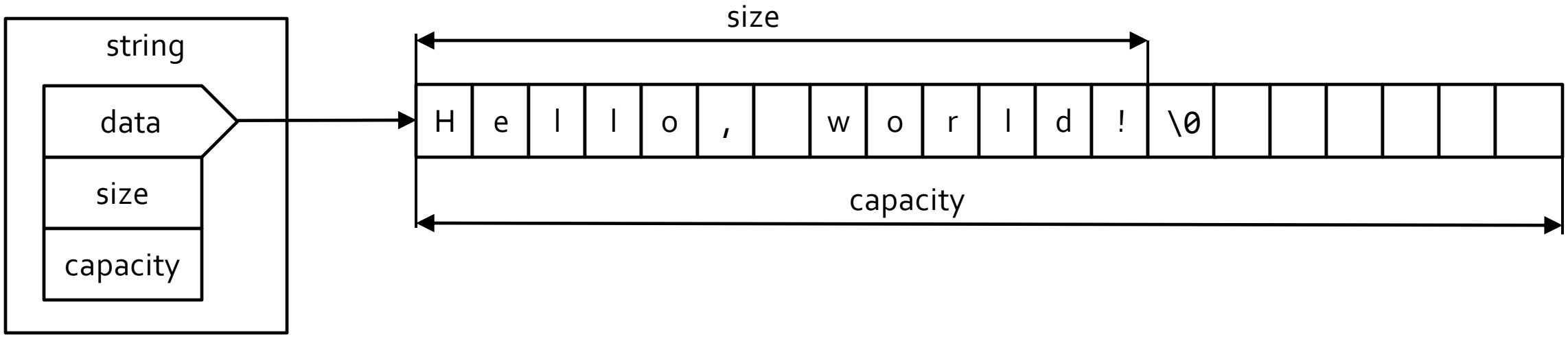
The Idea: Let's Write a String Class

- The real reason for the problems is that for C strings, the length is not an invariant.
- To preserve the invariants of objects such as strings, private state that is inaccessible for modification is needed, i.e., **encapsulation** is necessary.
- This naturally leads to the idea: write a string class.

A creative challenge

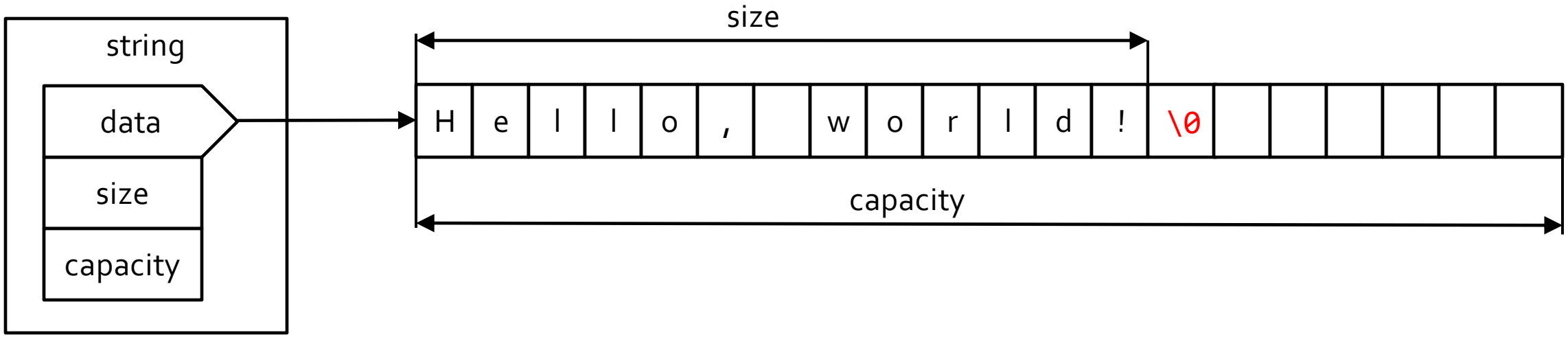
- Draw me a bicycle (and let's see if you can reinvent the wheel).
- Here are just a few of the many 'reinvented wheels' for strings that are actively used today
  - `CString (MFC / ATL)`
  - `QString (QT framework)`
  - `CComBSTR (ATL)`
  - `FBString (Facebook Folly)`
  - `GString (GTK Glib)`
  - `EASTL::string (EASTL)`
- Since you're unlikely to do any better, let's first take a look at how std::string is built.

How std::string is structured in principle



- This picture lies in one very significant detail.
- But it is good as a fundamental diagram.

How std::string is structured in principle



- The strangest thing in this picture seems to be the trailing null.
- Why is it needed if we **already** store the size?

https://godbolt.org/z/9fWcsY9c8

String as legacy string

- The `c_str()` method converts the string to const char pointer.
- The `data()` method converts the string to char pointer.

```
std::string s = "Hello, world!";
std::cout << s.c_str() << std::endl;
std::cout << s.data() << std::endl;
```

- By passing anything to a legacy API, you are somewhat compromising security.

std::string: A Net Positive

- Automatic memory management.

```
std::string s = "Hello"; // memory allocated and owned
```

- Size is a string invariant.
- A rich set of built-in methods and even its own literals.

```
auto sz = "Hello"s.find_first_of("klmn"s); // sz equals 2
```

- Compatibility with legacy C API.
- Have I sold you on standard strings? Any clouds on the horizon?

https://godbolt.org/z/jWbe7PhYx

Static strings

- What do you think about using constant static strings?

```
static const std::string s = "FOO";
// .....
int foo(const std::string &arg);
// .....
foo(s);
```

Memory allocation before main

- What do you think about using constant static strings?

```
static const std::string s = "FOO";

// .....

int foo(const std::string &arg);

// .....

foo(s);
```

- The idea looks bad: we are adding heap indirection. "FOO" is a literal. When the program is loaded, it will be copied to the heap.

https://godbolt.org/z/3Eq4ha5f9

Replacement with a pointer

- What do you think about replacing a static string with a pointer?

```
static const char *s = "FOO";
// .....
int foo(const std::string &arg);
// .....
foo(s);
```

Replacement with a pointer

- What do you think about replacing a static string with a pointer?

```
static const char *s = "FOO";
// .....
int foo(const std::string &arg);
// .....
foo(s);
```

- It got even worse: now we hit the creation of a temporary object on every call to the function foo.

https://godbolt.org/z/jzqTfd7qe

Solution: string_view (C++17)

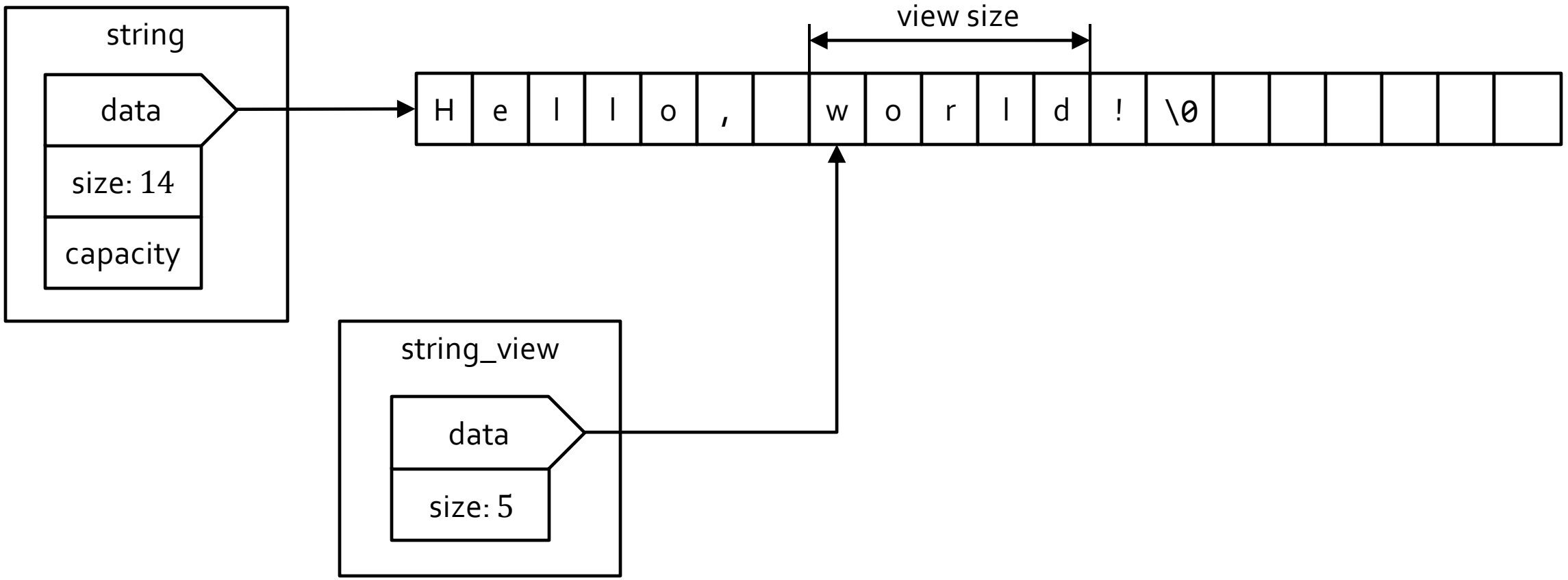
- `std::string_view` is a non-owning pointer to a string.

```
static constinit const std::string_view s = "FOO";
// .....
int foo(const std::string_view arg);
// .....
foo(s);
```

- There is neither heap indirection nor temporary object creation here.

https://godbolt.org/z/qqjMxjYab

How std::string_view is structured in principle



Let's give std::string_view a try

```
static constinit const std::string_view s = "  FOO  ";

std::string_view trim(std::string_view s) {
    auto start = s.find_first_not_of(" \t\n\r\f\v");
    if (start == std::string_view::npos)
      return "";
    auto end = s.find_last_not_of(" \t\n\r\f\v");
    return s.substr(start, end - start + 1);
}

std::println("[{}]", trim(s));
```


https://godbolt.org/z/9vGoqzd3K

Value-semantics

- What would you say about the following uses of string and string view?

```
const std::string& s1 = "hello world"; // 1
const std::string& s2 = std::string("hello world"); // 2
std::string_view sv1 = "hello world"; // 3
std::string_view sv2 = std::string("hello world"); // 4
```

Value-semantics

- What would you say about the following uses of string and string view?

```
const std::string& s1 = "hello world"; // OK
const std::string& s2 = std::string("hello world"); // OK
std::string_view sv1 = "hello world"; // OK
std::string_view sv2 = std::string("hello world"); // DANGLE
```

- The main problem with this kind of classes: they pretend to be values, but they are not.

One more dead parrot example

```
auto identity(std::string_view sv) { return sv; }
std::string s = "hello";
auto sv1 = identity(s);  // 1
auto sv2 = identity(s + " world");  // 2
```

One more dead parrot example

```
auto identity(std::string_view sv) { return sv; }

std::string s = "hello";

auto sv1 = identity(s);  // OK

auto sv2 = identity(s + " world");  // DANGLE
```

- Correct use of reference types: only as temporary values. They should not be stored.

A Rule of Thumb

- Entities with reference semantics should be used in two cases:
- In function parameters.

```
std::string identity(std::string_view sv) { return sv; }
```

- In for-loop initializers.

```
std::vector<std::string> elements;
// ...
for (std::string_view elt : elements)
  do_something(elt);
```

Do we see any other problems?

```
char astr[] = "hello";
char bstr[15];

int alen = strlen(astr);
assert(alen == 5);

strcpy(bstr, astr);
strcat(bstr, ", world!");
int res = strcmp(astr, bstr);
assert(res < 0);
```

```
std::string astr = "hello";
std::string bstr;
bstr.reserve(15);

int alen = astr.length();
assert(alen == 5);

bstr = astr;
bstr += ", world!";
int res = astr.compare(bstr);
assert(res < 0);
```

Memory allocations

```
char astr[] = "hello";
char bstr[15];

int alen = strlen(astr);
assert(alen == 5);

strcpy(bstr, astr);
strcat(bstr, ", world!");
int res = strcmp(astr, bstr);
assert(res < 0);
```

```
std::string astr = "hello";
std::string bstr;
bstr.reserve(15);

int alen = astr.length();
assert(alen == 5);

bstr = astr;
bstr += ", world!";
int res = astr.compare(bstr);
assert(res < 0);
```

Forming strings

- Direct concatenation.

```
std::string result, proto = ssl ? "https" : "http";
result = proto + "://" + path + "/" + query;
```

- Input-output streams.

```
std::stringstream ss;
ss << proto << "://" << path << "/" << query;
result = ss.str();
```

- Formatting.

```
result = std::format("{}://{}/{}", a, path, query);
```


benchmarking

Not All Benchmarks Are Created Equal

```
for (auto _ : state) {
  std::stringstream ss;
  ss << (ssl ? "https" : "http") << "://" << path << "/" << query;
  auto s = ss.str();
  benchmark::DoNotOptimize(s);
}
```

```
std::stringstream ss;
for (auto _ : state) {
  if (ss.rdbuf()) ss.rdbuf()->pubseekpos(0);
  ss << (ssl ? "https" : "http") << "://" << path << "/" << query;
  auto s = ss.str();
  benchmark::DoNotOptimize(s);
}
```


benchmarking

Structure of <format>

- Starting from C++20, std::format is defined as:

```
template <typename... Args>
std::string format(std::format_string<Args...> fmt,
                   Args&&... args);
```

- Here `std::format_string` is a wrapper on a top of `std::string_view`.

```
std::string vformat(std::string_view sfmt,
                    std::format_args fargs);
```

- It is possible to rewrite format as vformat.

```
std::vformat(fmt.get(), std::make_format_args(args...));
```

Discussion

- Using std::print and std::format instead of I/O streams is still not an obvious solution.
- At the very least, you again have to parse the format string.

```
std::println("{} {:7} {}", j, i, j);
std::println("{} {:<7} {}", j, i, j);
std::println("{} {:_>7} {}", j, i, j);
std::println("{} {:_^7} {}", j, i, j);
```

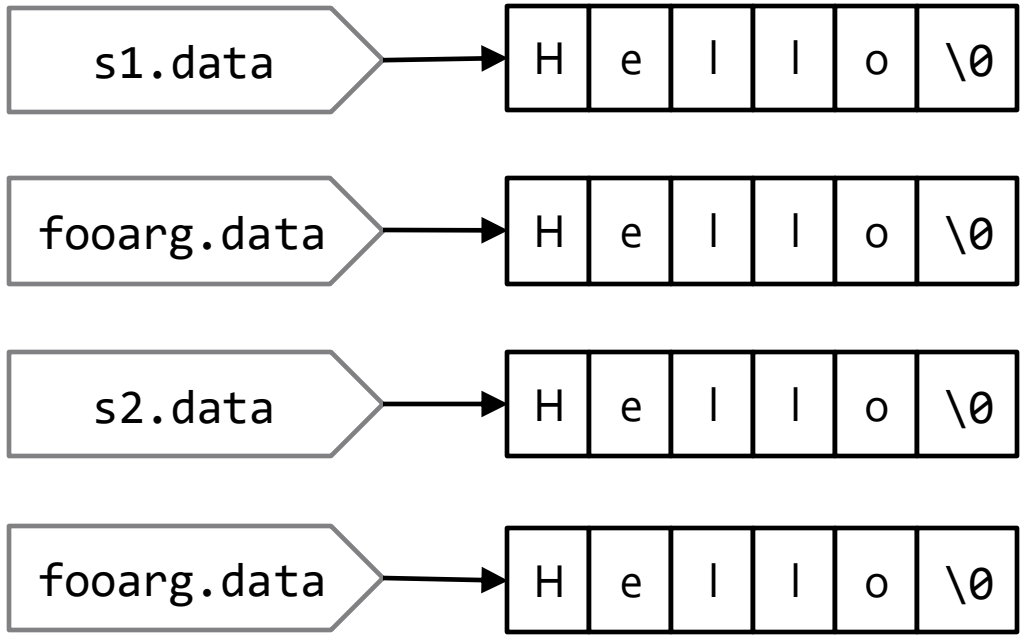
- On the other hand, it works for constexpr context.

https://godbolt.org/z/de3o9jv6K

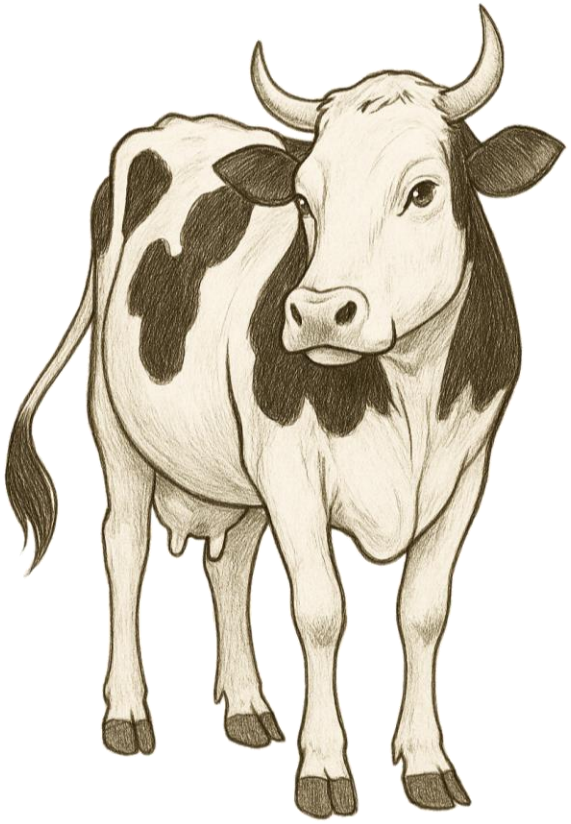
A bit more about performance

- Very often, dozens of copies of the same string live simultaneously in a program.

```
void foo(string s);
std::string s1 = "Hello";
foo(s1);
std::string s2 = s1;
foo(s2);
```



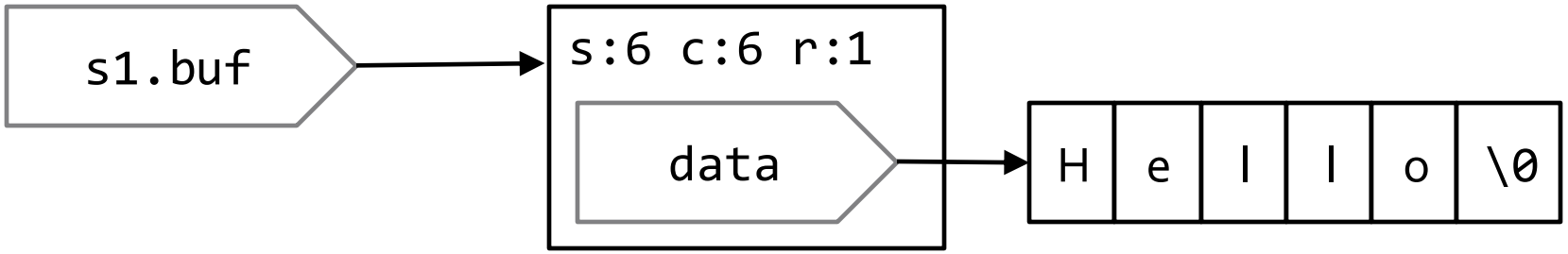
Copy On Write (COW)



- What if we try to count references in a string?

```
class stringbuf {
  char *data;
  size_t size;
  size_t capacity;
  int refcount;
// etc ....
class string {
  stringbuf *buf;
// etc ....
```

```
string s1 = "Hello";
string s2 = s1;
```

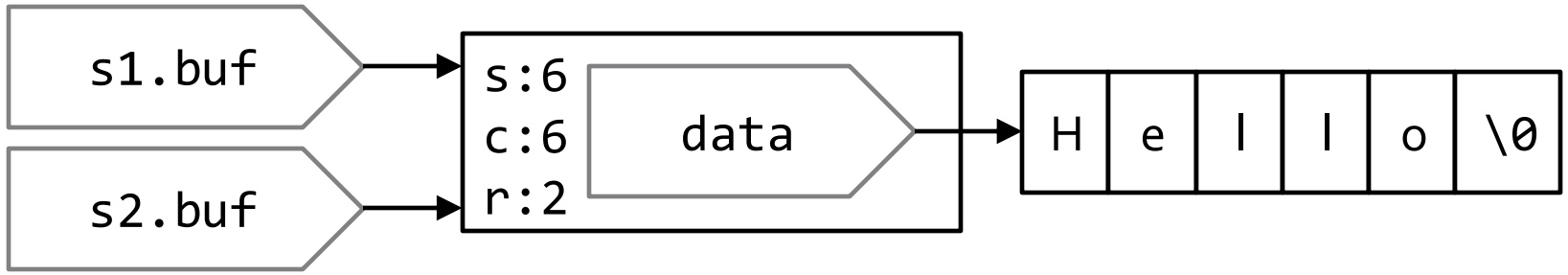


Copy On Write (COW)

- What if we try to count references in a string?

```
class stringbuf {
  char *data;
  size_t size;
  size_t capacity;
  int refcount;
// etc ....

class string {
  stringbuf *buf;
// etc ....
```

```
string s1 = "Hello";
string s2 = s1;
```

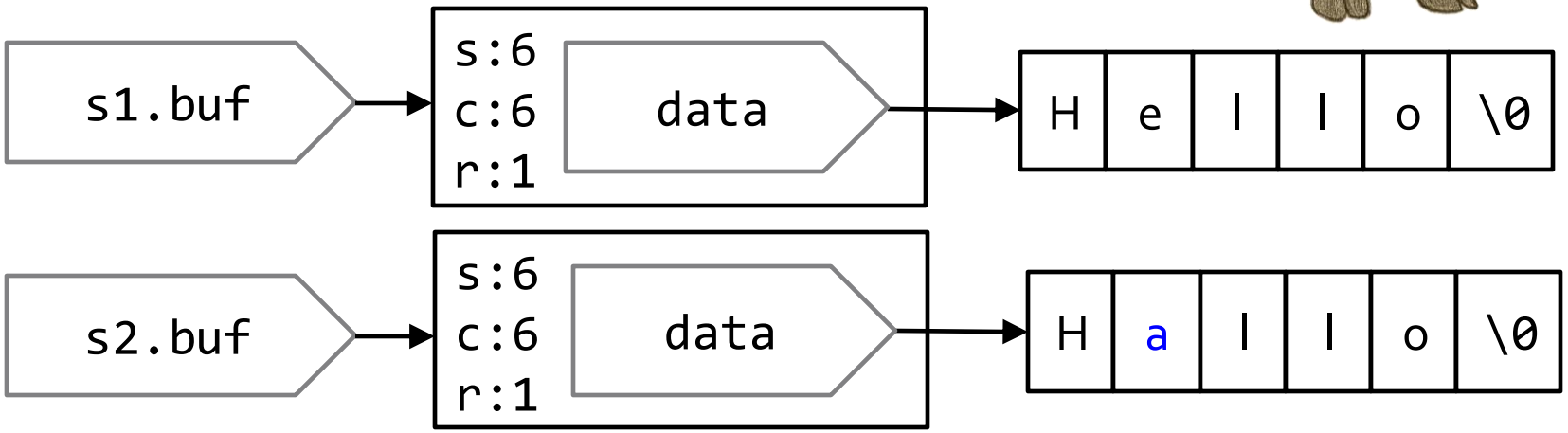


Copy On Write (COW)

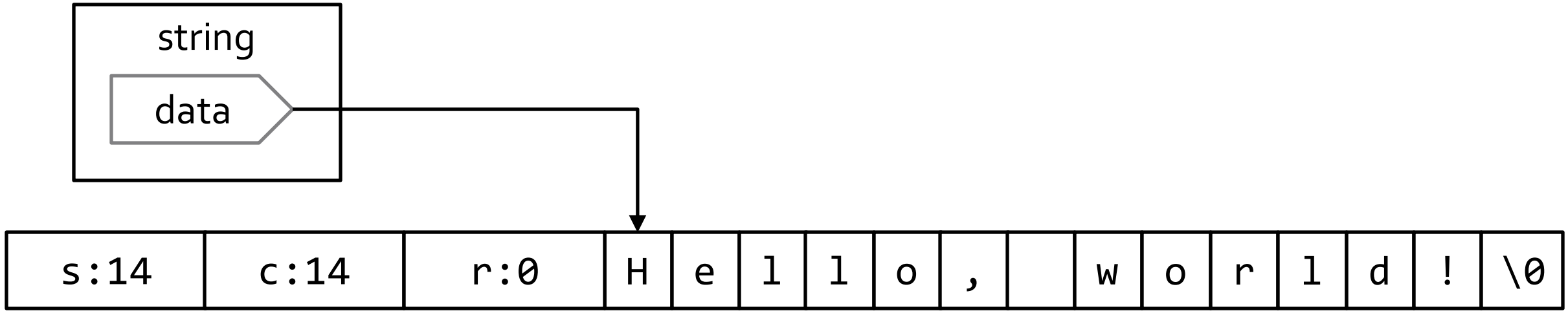
- What if we try to count references in a string?

```
class stringbuf {
  char *data;
  size_t size;
  size_t capacity;
  int refcount;
// etc ....

class string {
  stringbuf *buf;
// etc ....
```

```
string s1 = "Hello";

string s2 = s1;
s2[1] = 'a';
```



GCC string (version < 5), libstdc++



- The reference count is stored as -1, so it is zero in the picture.
- COW is actively used.

Discussion

- From the very beginning, the COW idiom had its supporters and opponents.
- Which side are you on?

Advantages and disadvantages

- From the very beginning, the COW idiom had its supporters and opponents.
- Memory savings.
- Cheap copying (just incrementing the reference count).
- Fewer allocations and deletions in the heap => performance gain.
- Extra level of indirection.
- The copy operation virally spreads into all modifying operations.
- Thread safety issues (Multithread COW disease).
- However, there is a consideration that breaks the balance. This is pointer invalidation.

Pointer Invalidation

- Operations on a string can invalidate pointers into the string. For example:

```
std::string a = "Hello";
const char *p = &a[3];
a += "world"; // beyond this point, p cannot be used
```

- Here, there is no problem.
- The problem is that with COW, pointers are invalidated by seemingly harmless operations.

Pointer Invalidation

- Operations on a string can invalidate pointers into the string. For example:

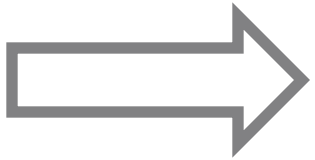
```
std::string s("str");
const char* p = s.data();

{
  std::string s2(s);
  s[0] = 'S';
}

std::cout << *p << '\n';
```

- For non-COW strings, p is still valid, but for COW it may no longer be.

Pointer Invalidation

- In 2011, it was officially forbidden to invalidate pointers when executing operator[] (C++11, [string.require]).
- This excludes COW implementations of `std::string`.
- Desired outcome: COW is (almost) dead.
- In reality, the removal of COW strings from the standard without introducing a worthy replacement spawned a bunch of bicycles (e. g. QString).

COW is (almost) dead

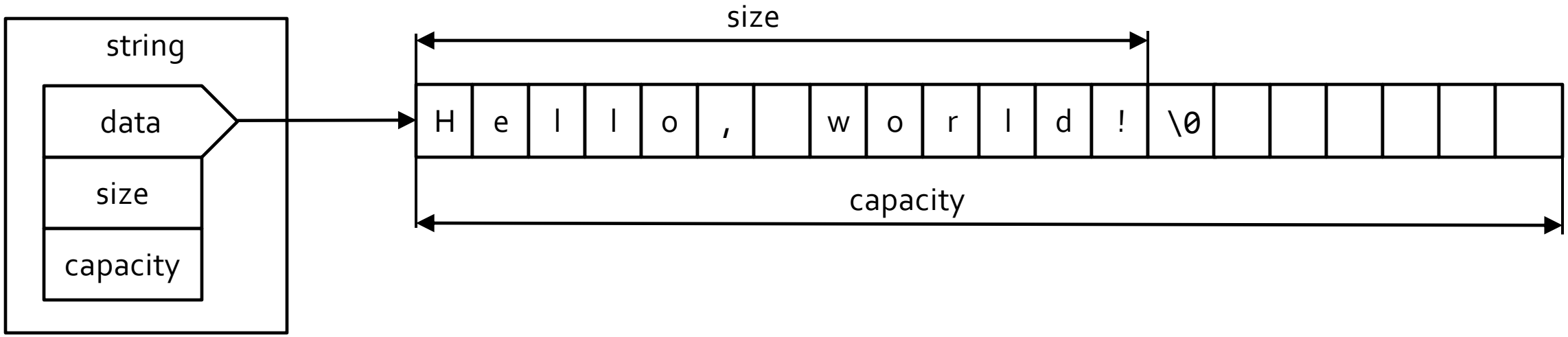


Current state

Desired state

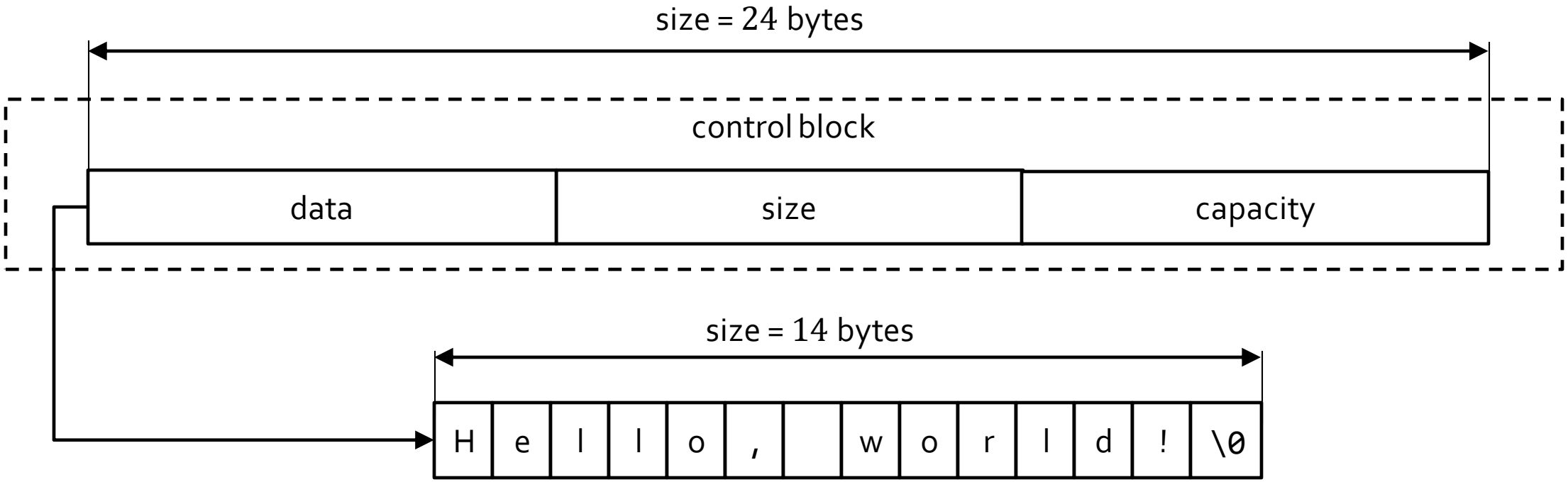
- But does this mean that we can't do anything at the class design level?

Discussion: let's return to the picture



- This picture is incorrect in one very significant detail
- Guess what is wrong?

Same picture, correct scale



- We may easily fit small data in the control block.

Small string optimizations (SSO)

- We don't need allocations for small strings.

```
class string {
  size_type size_;
  union {
    struct {
      char *data_;
      size_type capacity_;
    } large_;
    char small_[sizeof(large_)];
  };
// and so on
```

Discussion

- Any problems with this approach?

```
class string {
  size_type size_;
  union {
    struct {
      char *data_;
      size_type capacity_;
    } large_;
    char small_[sizeof(large_)];
  };
// and so on
```

Yes, there are some problems

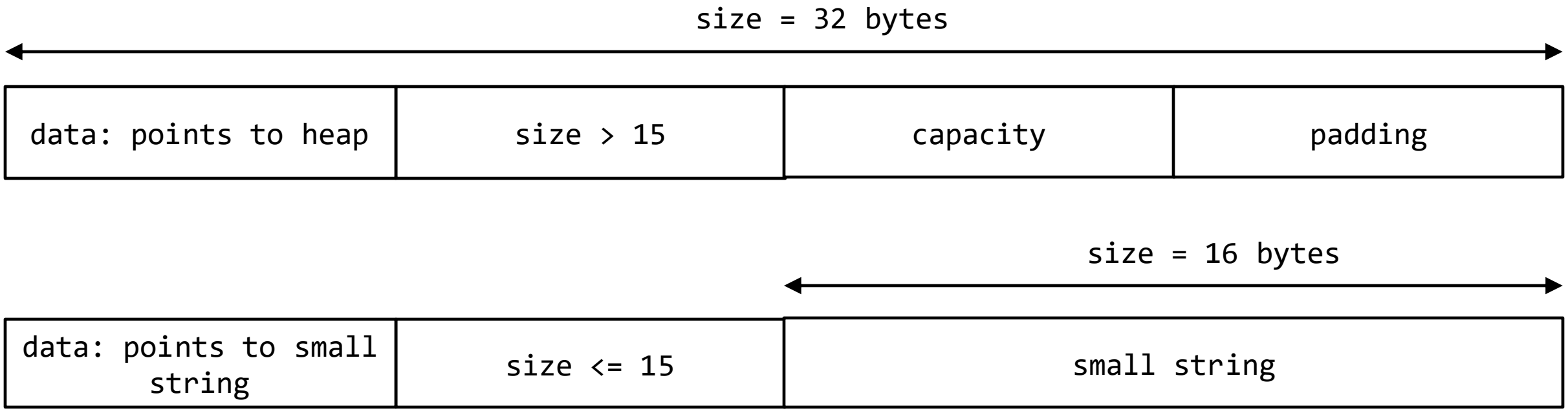
- What disadvantages do you see in this approach to SSO?
- Copying becomes more complicated.
- Moving becomes non-trivial.
- Selection time is added.

```
this->small_[i]
```

```
this->large_.data[i]
```

- You are paying for every access (including reading) with a size check.
- The latter problem is more serious. Can anything be done about it?

GCC string (version >= 5), libstdc++



- We traded off SSO buffer size for faster string operations.

Problem: now let's consider UTF32

- In the case where one character does not occupy one byte (but, for example, four), SSO has problems.
- But first of all, we have problems. How to generalize the developed string for characters of different sizes?
- First idea: let's write three separate classes for `utf8string`, `utf16string`, and `utf32string`.
- I'd like to get your feedback on this idea.

String class template

- How `basic_string` is structured **in principle**.

```
template <typename CharT> class basic_string {
  CharT *data;
  size_t size;

  union {
    size_t capacity;
    enum {SZ = (sizeof(data) + 2 * sizeof(size_t) + 31) / 32};
    CharT small_str[SZ];
  } sso;

public:
  // all 89 methods here
};
```

Using for convenience

```
using string = basic_string<char>;
using u16string = basic_string<u16char_t>;
using u32string = basic_string<u32char_t>;
using wstring = basic_string<wchar_t>;
```

- Now we have separate typedefs and single generic class.

Type traits

- There are many questions, the answers to which are different for different strings with different character types.
- It is reasonable to combine all this into a class

```
template <typename CharT> class char_traits;
```

- With methods like `assign, eq, lt, move, compare, find, eof, ....`

```
template <typename CharT,
          typename Traits = std::char_traits<CharT>>
class basic_string {

// all we are doing we are doing via traits
```

Discussion

- Is the memory allocation method per character a trait of the character?

Allocators

- Abstract away memory allocation (where to go to get some memory).

```
template <typename CharT,
          typename Traits = std::char_traits<CharT>
          typename Allocator = std::allocator<CharT>>

class basic_string {

// all we are doing we are doing via traits and allocator
```

Strings not only for symbols

- The following code is ill-formed

```
void toggle(std::vector<bool>& bits) {
  for (auto &b : bits)
    b = !b;
}
```

- What can we do?

https://godbolt.org/z/73MTKMr10

Strings not only for symbols

- Let's use basic string!

```
void toggle(std::basic_string<bool>& bits) {
  for (auto &b : bits)
    b = !b;
}
```

- We would prefer basic_string_view but it is immutable.

span approach

```
void toggle(std::span<bool> bits) {
  for (auto &b : bits)
    b = !b;
}

int main() {
  auto osit = std::ostream_iterator<bool>(std::cout, " ");
  std::basic_string<bool> v = {1, 0, 0, 1, 1};
  toggle(v);
  std::copy(v.begin(), v.end(), osit);
  std::cout << std::endl;
}
```


https://godbolt.org/z/WjdWWbv71

Discussion

- Are you ready to invent some wheels now?

Homework assignment

- $[HW2.1][3]$ Write your own great COW-string template class. Measure advantage over std::string on some benchmarks.
- $[HW2.2][3]$ Write template class `string_twine` for $O(\log(N))$ concatenation of several `string_view's`.

```
std::string_view sv = "Hello,", sv2 = "World!";

auto s = string_twine(sv, " ", sv2).str(); // -> string
```

- $[HW2.3][2]$ Compilers now optimizing std::string worse then std::vector. Investigate what is happening: https://godbolt.org/z/Tfh6zfa6P

Bibliography

- ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC 14882:2023
- Bjarne Stroustrup, The C++ Programming Language (4th Edition), 2013
- Nicholas Ormrod, The strange details of std::string at Facebook, CppCon, 2016
- Антон Полухин, Как делать не надо: C++ велосипедостроение для профессионалов, C++ Russia, 2017
- Victor Ciura, Enough string_view to Hang Ourselves, CppCon, 2018
- Brian Ruth, std::basic_string: for more than just text, CppCon, 2018
- Marc Gregoire, C++20 String Formatting Library: An Overview and Use with Custom Types, CppCon, 2020
- Jonathan Müller, C++ String Literals Have the Wrong Type, C++ on Sea, 2023