СТРОКИ

Работа со строками как мотивирующий пример обобщённого программирования.

К. Владимиров, Syntacore, 2024
mail-to: konstantin.vladimirov@gmail.com

Hello, world!

```
import std;

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

```
<source>:2:8: fatal error: module 'std' not found
    2 | import std;
      |        ^~~
1 error generated.
```


https://godbolt.org/z/5eY3n7q4M

Hello, world!

```
#include <print>

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

```
> g++ --std=c++23 -O2 hello.cc

> ./a.out

Hello, world!
```


https://godbolt.org/z/1xa4sbe6f

Hello, world!

```
#include <iostream>

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

> g++ -O2 hello.cc

> ./a.out

Hello, world!
```


https://godbolt.org/z/M3K405Wza

Что такое строка?

```
std::cout << "Hello, world!" << std::endl;
```

- Это строка?

Что такое строка?

```
std::cout << "Hello, world!" << std::endl;
```

- Это строковый литерал.
- Примеры строковых литералов: `u8"Hello"`, `u"Hello"`, `U"Hello"`, `L"Hello"`

```
std::cout << R"(h
e
l
l
o)" << std::endl;
```

- Примеры других литералов: `0x1000`, `1.6e-2`, `'c'`, `true`, `nullptr`

Типы литералов

- `0x1000 // → int`
- `1.6e-2 // → double`
- `'c' // → char`
- `true // → bool`
- `nullptr // → nullptr_t`
- `"Hello" // → ???`
- Мы точно знаем, что:


```
auto PHello = "Hello"; // → const char*
```

Или всё таки указатель?

```
auto t = "Hello, world!"; // → const char *
std::cout << sizeof("Hello, world!") << std::endl; // → 14
std::cout << sizeof(t) << std::endl; // → 8
std::cout << "Hello, world!" + 7 << std::endl; // → world
std::cout << 7["Hello, world!"] << std::endl; // → w
```

- Итак, что же особенного в строковых литералах?
- Дело в том, что они как выражения всегда **glvalue**, а остальные литералы всегда **prvalue**.

https://godbolt.org/z/PYsbbKcM8

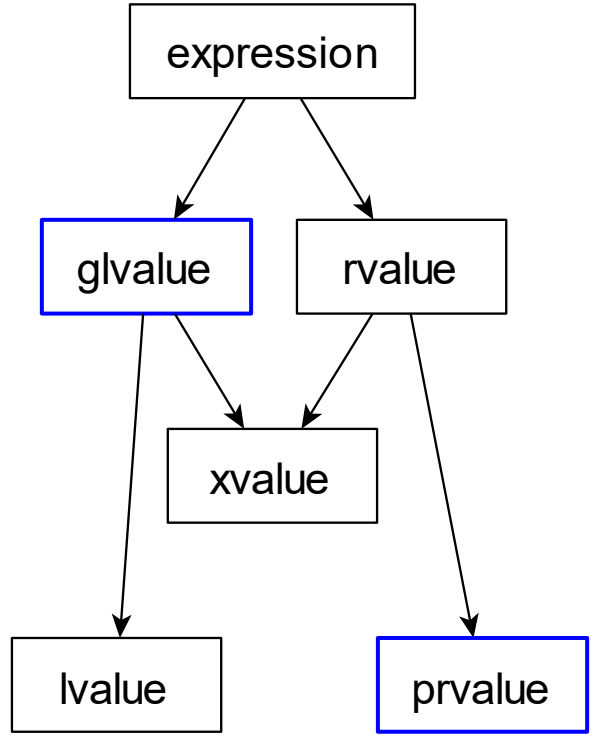
Выражения

- An expression is a sequence of operators and operands that specifies a computation.

```
int a = 5, b;

b = a + 2; // a + 2 это prvalue expression
```

- glvalue это выражение, идентифицирующее (истекающий, но) постоянный объект.
- prvalue это рецепт для создания объекта.
- Whenever a glvalue appears as an operand of an operator that expects a prvalue for that operand, [...] standard conversions are applied to convert the expression to a prvalue **[basic.lval]**.



Литералы как выражения

- Строковый литерал это всегда lvalue. Остальные литералы это всегда prvalue (кроме пользовательских, там сложнее).

```
int a = 5, b;

b = a + 2; // lvalue to rvalue

auto p = "Hello" + 3; // array to pointer
```

- Кроме array to pointer есть ещё неявное преобразование function to function pointer.

```
int f(int x);

auto pf = f; // function to function pointer
```

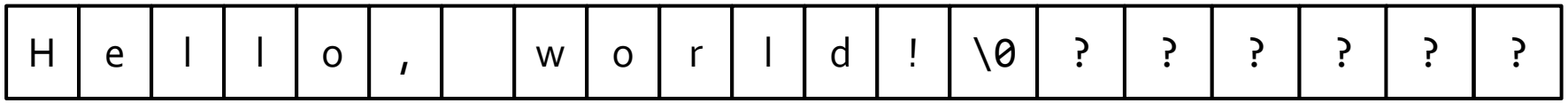
Строка это (указатель на) массив chars?

- Допустим мы завели такой массив.

```
char Hello[20];
strcpy(Hello, "Hello, world!");
std::cout << Hello << std::endl;
```

- Простой вопрос. Что вы думаете про использование print, будет ли разница?

```
std::println("{0}", Hello); // ?
```



Строка это (указатель на) массив chars?

- Допустим мы завели такой массив.

```
char Hello[20];
strcpy(Hello, "Hello, world!");
std::cout << Hello << std::endl; // ok
```

- Простой вопрос. Что вы думаете про использование print, будет ли разница?

```
std::println("{0}", Hello); // garbage out?
std::println("{0}", +Hello); // ok for sure
```

- Вероятно println просто требует позитивного отношения!

Почему я вообще сомневаюсь?

- У `std::print` как и у `std::format` просто нет decaying point.

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

- Array-to-pointer преобразование случается при вычислении выражения.

```
char Hello[20];

strcpy(Hello, "Hello, world!");

std::println("{0}", Hello); // Pack: fmt, char[20]&&

std::println("{0}", +Hello); // Pack: fmt, char*&&
```

Почему я могу передать его в strlen?

```
std::size_t strlen(const char* str);

std::size_t mystrlen(const char** str) {
  return strlen(*str);
}

char Hello[20]; strcpy(Hello, "Hello, world!");

std::cout << strlen(Hello) << std::endl; // → 13

std::cout << mystrlen(&Hello) << std::endl; // → ERROR
```

- Казалось бы ситуация симметричная, но то работает, то нет.
- Для меня лично несколько более странно, что работает…

https://godbolt.org/z/5T5TW4TjY

Qualification conversion

```
char *cPtr;
const char ** cA = &cPtr; // → ERROR
const char * const * cB = &cPtr; // → OK
```

- И возвращаясь к примеру с mystrlen, см. [conv.qual].

```
std::size_t mystrlen(const char* const* str);
char Hello[20]; char *pHello = Hello;
mystrlen(&Hello); // → ERROR
mystrlen(&pHello); // → OK
```


https://godbolt.org/z/sqoKW3nPY

Работа с C-строками: <cstring>

- 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);
```

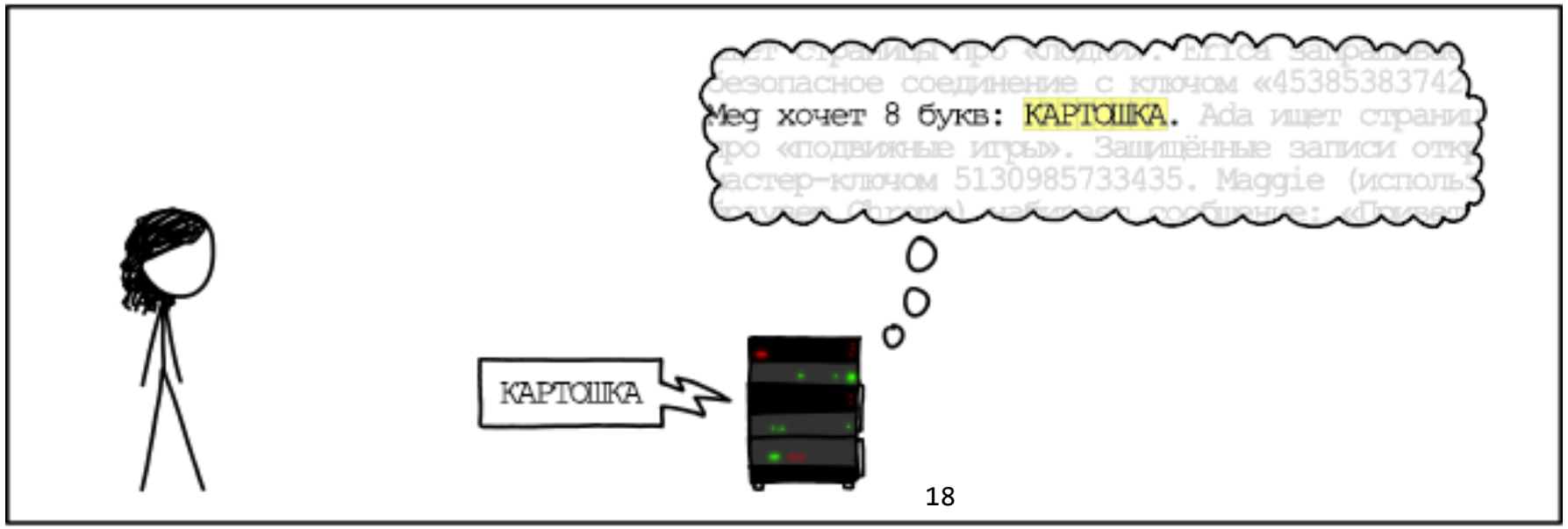
Обсуждение

- При передаче в функцию вроде strcpy указателя на неправильные данные, мы получим в ответ все данные до ближайшего нулевого символа.
- Вариант решения в стиле C: функции с ограничением количества символов.

```
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);
```

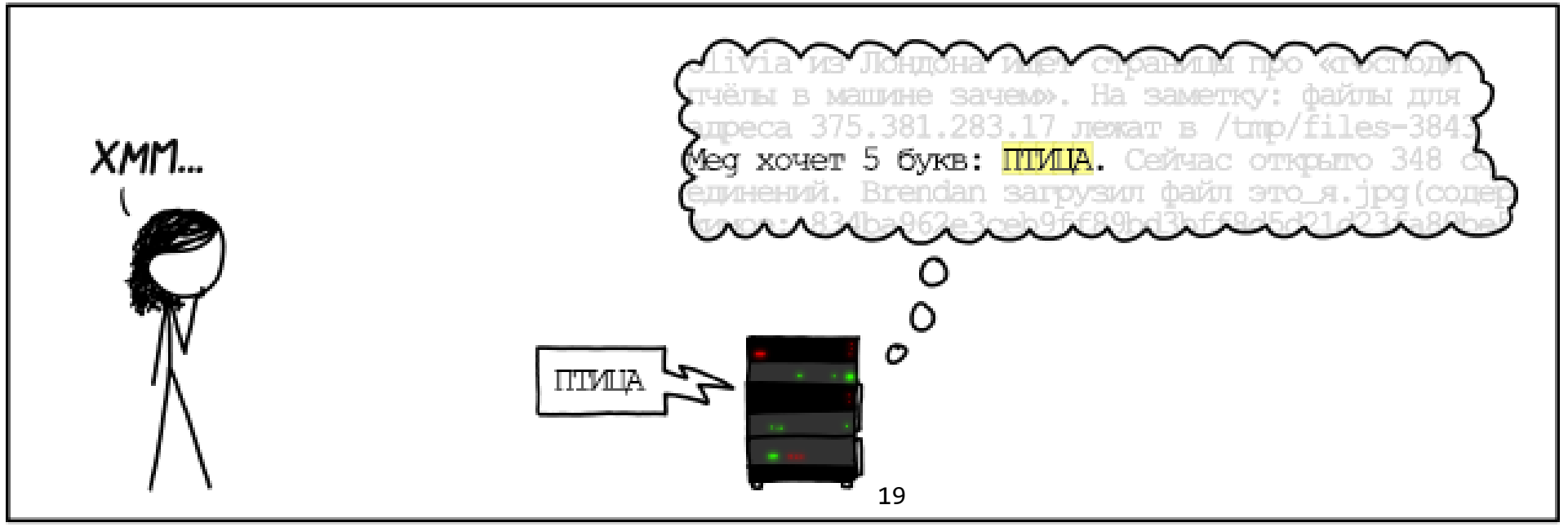
- Работает ли этот вариант?

СУТЬ УЯЗВИМОСТИ HEARTBLEED:



xkcd.ru/1354/


СЕРВЕР, ТЫ ЗДЕСЬ? ЕСЛИ ДА, СКАЖИ «ПТИЦА» (5 БУКВ).
Meg хочет 5 букв: ПТИЦА.


ХММ...
Meg хочет 5 букв: ПТИЦА.
ПТИЦА

xkcd.ru/1354/


СЕРВЕР, ТЫ ЗДЕСЬ? ЕСЛИ ДА, СКАЖИ «ШЛЯПА» (500 БУКВ).
Meg хочет 500 букв: ШЛЯПА. Lucas запросил стр
ницу «знакомства в метро». Eve (админ) хочет
поставить на сервер мастер-ключ «14835038534
Isabel ищет страницы про «змеи но не слишком
длинные». Пользователь Karen меняет свой паро
на «ПрЛоВаСю». Пользователь Amber ищет стр
Meg хочет 500 букв: ШЛЯПА. Lucas запросил стр
ницу «знакомства в метро». Eve (админ) хочет
поставить на сервер мастер-ключ «14835038534»
Isabel ищет страницы про «змеи но не слишком
длинные». Пользователь Karen меняет свой паро
на «ПрЛоВаСю». Пользователь Amber ищет стр
ШЛЯПА. Lucas запросил страницу «знако
ства в метро». Eve (админ) хочет пост
вить на сервер мастер-ключ «148350385
4». Isabel ищет страницы про «змеи но
е слишком длинные». Пользователь Karen
меняет свой пароль на «ПрЛоВаСю». Поль

xkcd.ru/1354/

Обсуждение

- Настоящая причина проблем – в том, что для C строки длина не является инвариантом.

Обсуждение

- Настоящая причина проблем – в том, что для C строки длина не является инвариантом.
- Чтобы сохранять инварианты таких объектов как строки, необходимо закрытое состояние, недоступное к модификации, т.е. необходима инкапсуляция.
- Что естественным образом приводит к идее: написать класс строки.

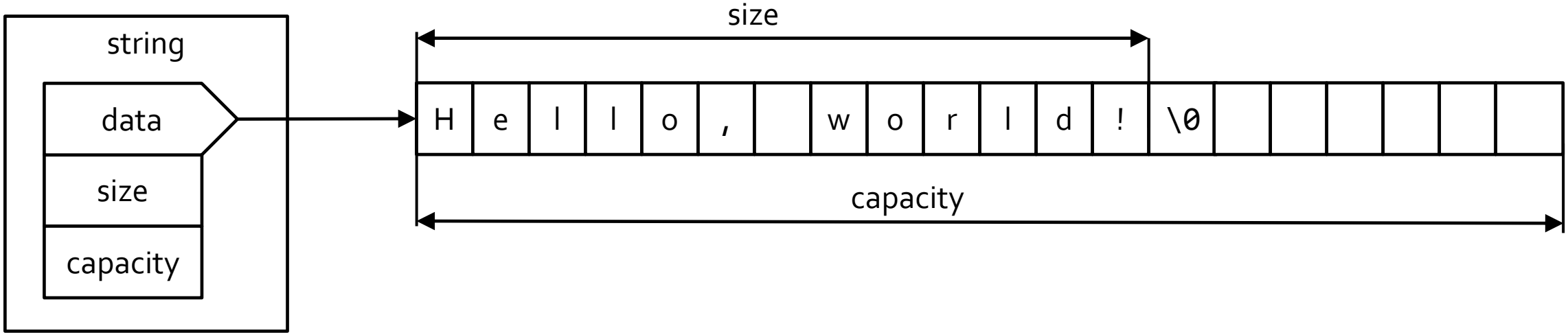
Творческая задача

- Нарисуйте на листочке бумажки велосипед.

Творческая задача

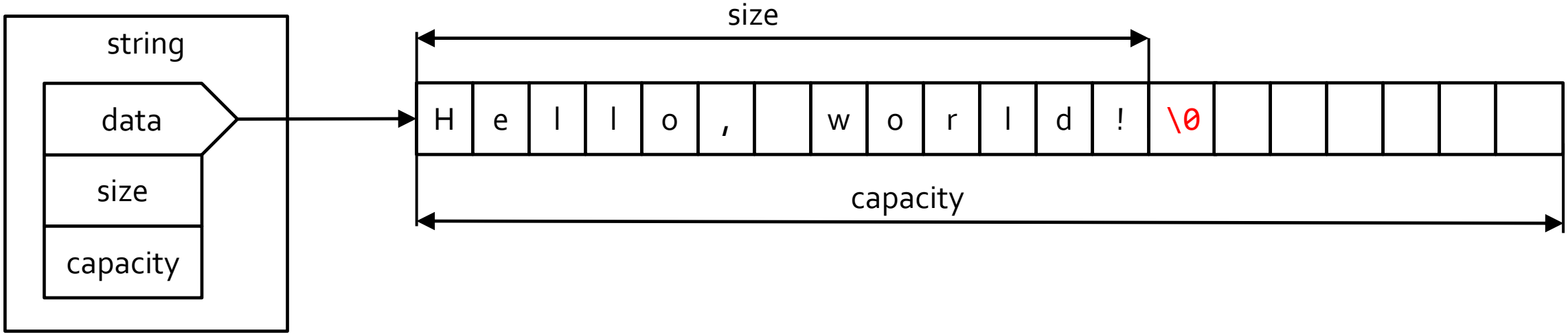
- Нарисуйте на листочке бумажки велосипед.
- Вот только некоторые из существующих и активно используемых велосипедов для строк:
  - `CString`
  - `QString`
  - `CComBSTR`
  - `FBString`
- Поскольку вы всё равно вряд ли сделаете лучше, давайте сначала посмотрим как устроен класс `std::string`.

Как в принципе устроен std::string



- Эта картинка врёт в одной очень существенной детали.
- Но она хороша как принципиальная схема.

Как в принципе устроен std::string



- Самым странным в этой картинке кажется завершающий ноль.
- Зачем он нужен, если мы уже храним размер?

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

Строка как легаси строка

- Метод `c_str()` приводит строку к `const char*`.
- Метод `data()` приводит строку к `char*`.

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

- Почему не implicit cast?
- Или вопрос надо задать иначе: почему эти методы вообще существуют?

Всё бывает очень плохо

```
float arr[1024];
memcpy(arr, s.c_str(), sizeof(float) * 1024);
```

- Относитесь к c_str() как к своего рода reinterpret_cast.

Обсуждение

- Стандартные строки достаточно хороши?

Обсуждение

- Стандартные строки достаточно хороши?
- Кажется они имеют ряд проблем по сравнению с C-строками

Проблема #1: статические строки

- Что вы думаете об использовании константных статических строк?

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

Статические строки

- Что вы думаете об использовании константных статических строк?

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

- Идея выглядит плохой: мы добавляем heap indirection. "FOO" это литерал. При загрузке программы он будет скопирован в кучу.

Замена указателем

- Что вы думаете о замене статической строки указателем?

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

Замена указателем

- Что вы думаете о замене статической строки указателем?

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

- Стало ещё хуже: теперь мы попадаем на создание временного объекта при каждом вызове функции foo.

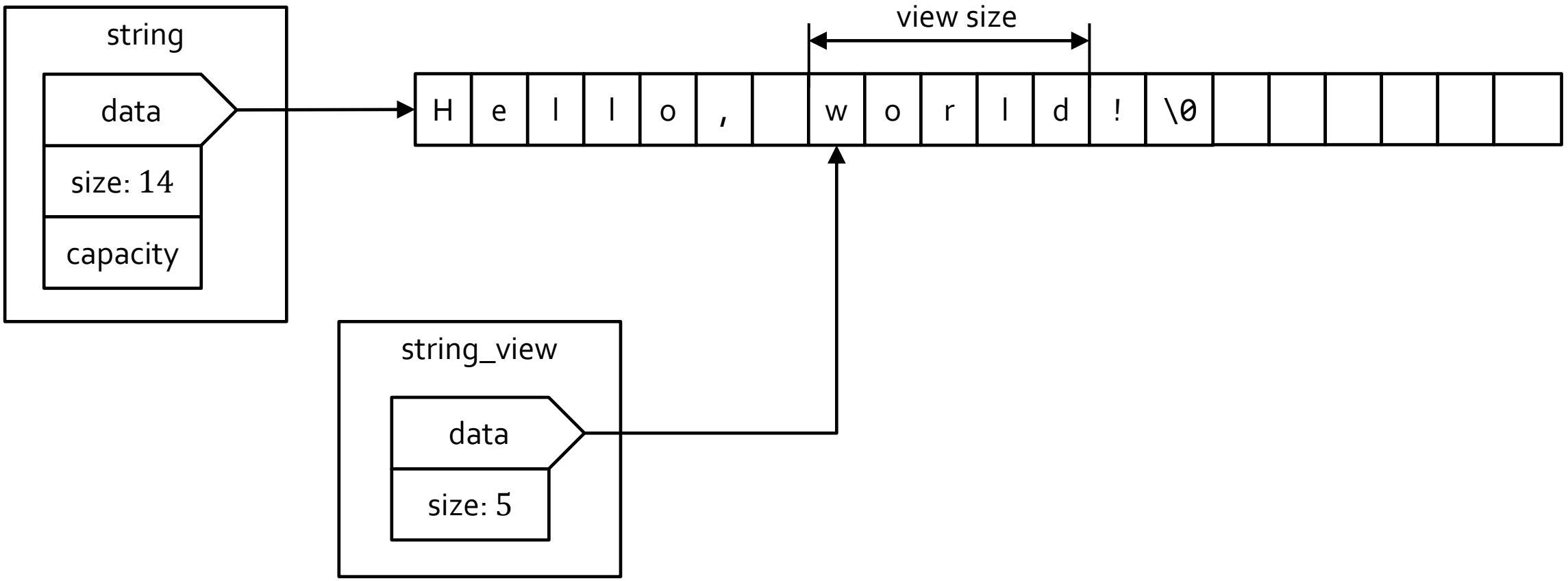
Решение: string_view (C++17)

- `std::string_view` это невладеющий указатель на строку.

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

- Здесь нет ни heap indirection ни создания временного объекта.

Как в принципе устроен std::string_view



Базовые операции над string_view

- remove_prefix
- remove_suffix
- copy
- substr
- compare
- find
- data

```
std::string str = "   trim me  ";
std::string_view vtrim = str;
auto trimfst = vtrim.find_first_not_of(" ");
vtrim.remove_prefix(std::min(trimfst,
                             vtrim.size()));
auto trimlst = vtrim.find_last_not_of(" ");
vtrim.remove_suffix(
  vtrim.size() - std::min(trimlst,
                          vtrim.size()));
```

Value-семантика

- Что вы скажете про следующие использования string и 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-семантика

- Что вы скажете про следующие использования string и 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
```

- Основная проблема такого рода классов: они притворяются значениями, но не являются ими.

Ещё немного примеров

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

Ещё немного примеров

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

std::string s = "hello";

auto sv1 = identity(s);  // OK

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

- Корректное использование ссылочных типов: только как временные значения. Их не следует сохранять.

Простое правило

- Сущности со ссылочной семантикой должны использоваться в двух случаях.
- В параметрах функций.

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

- В for-loop инициализаторах.

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

Проблема #2: выделения памяти

```
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);
```

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

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

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

Формируем строки

- Прямое сложение.

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

- Потоки ввода-вывода.

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

- Форматирование.

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

Замеры, впрочем, бывают разные

```
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);
}
```

Устройство format

- Начиная с C++20, `std::format` определён как:

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

- Здесь `std::format_string` это обёртка на `std::string_view`.

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

- Можно переписать format как vformat.

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

Обсуждение

- Использование `std::print` и `std::format` вместо потоков ввода-вывода до сих пор является не очевидным решением.
- Как минимум у вас снова появляется парсинг форматной строки.

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

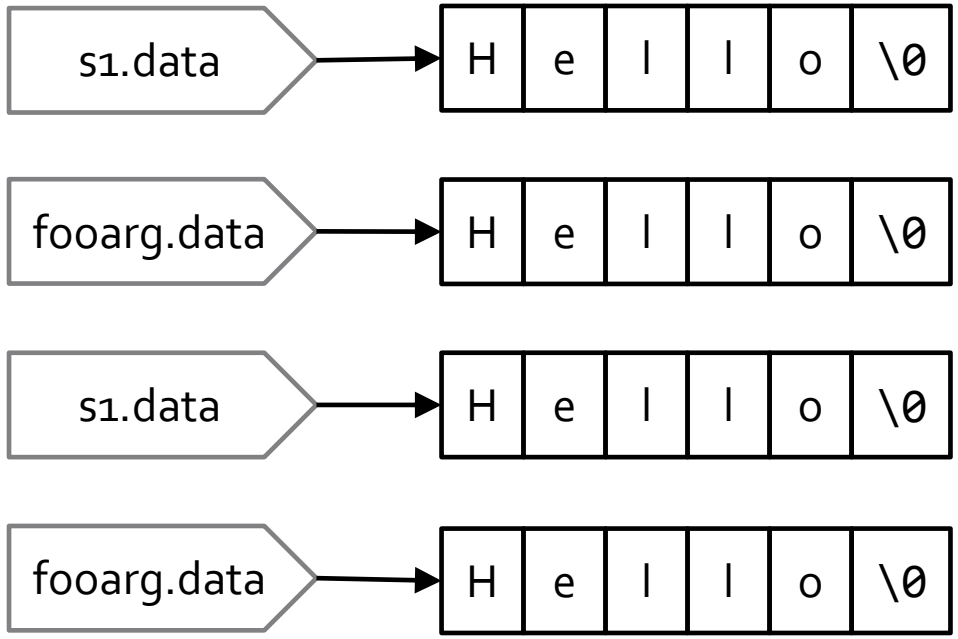
- С другой стороны зато это работает для constexpr контекста.

https://godbolt.org/z/de3o9jv6K

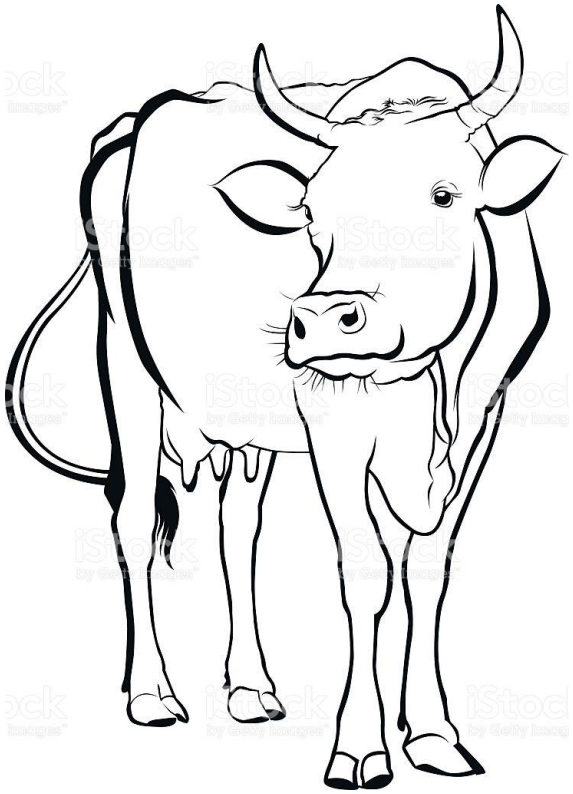
Ещё немного о производительности

- Очень часто в программе одновременно живут десятки копий одной и той же строки.

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



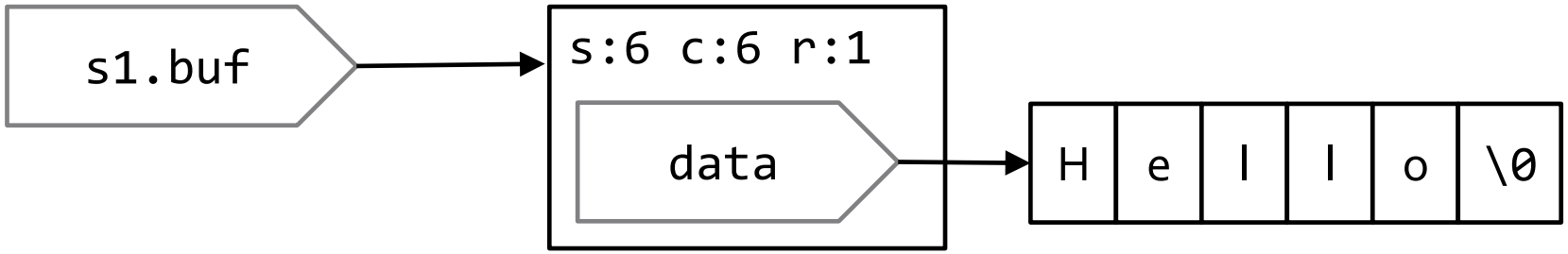
Copy On Write (идиома COW)



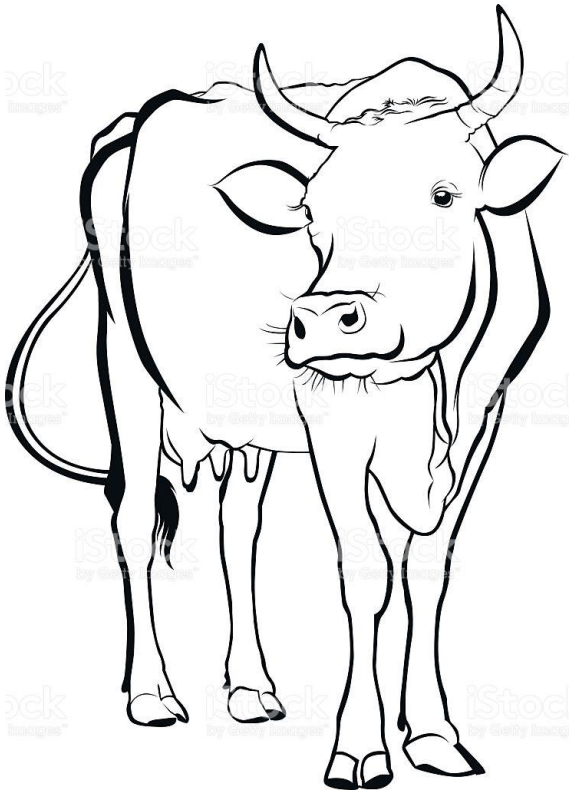
- Что если попробовать считать ссылки в строке?

```
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 (идиома COW)

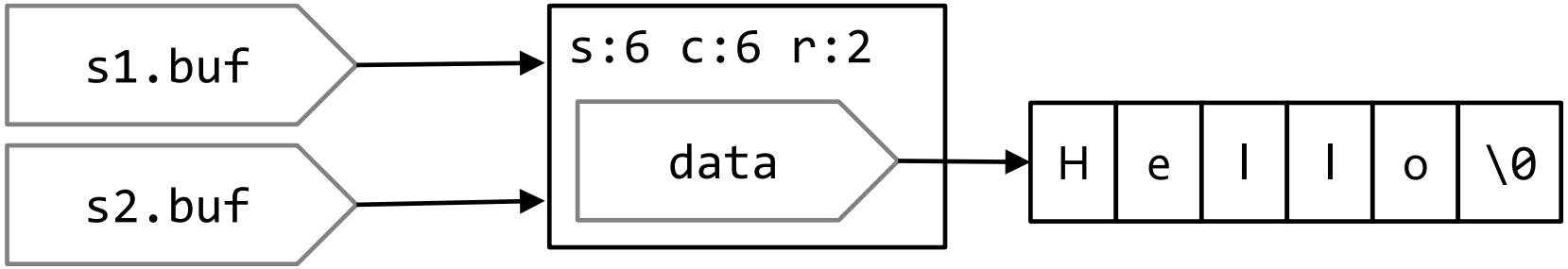


- Что если попробовать считать ссылки в строке?

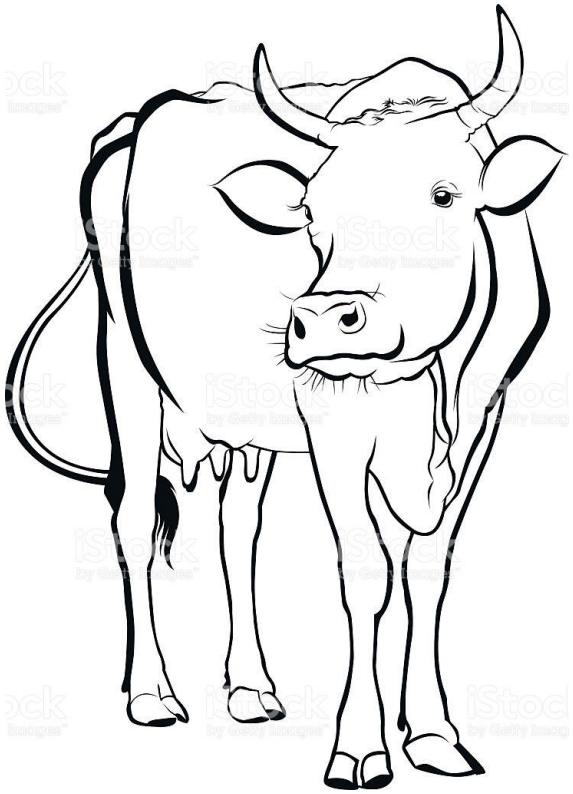
```
class stringbuf {
   char *data;
   size_t size;
   size_t capacity;
   int refcount;
// etc ....

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

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



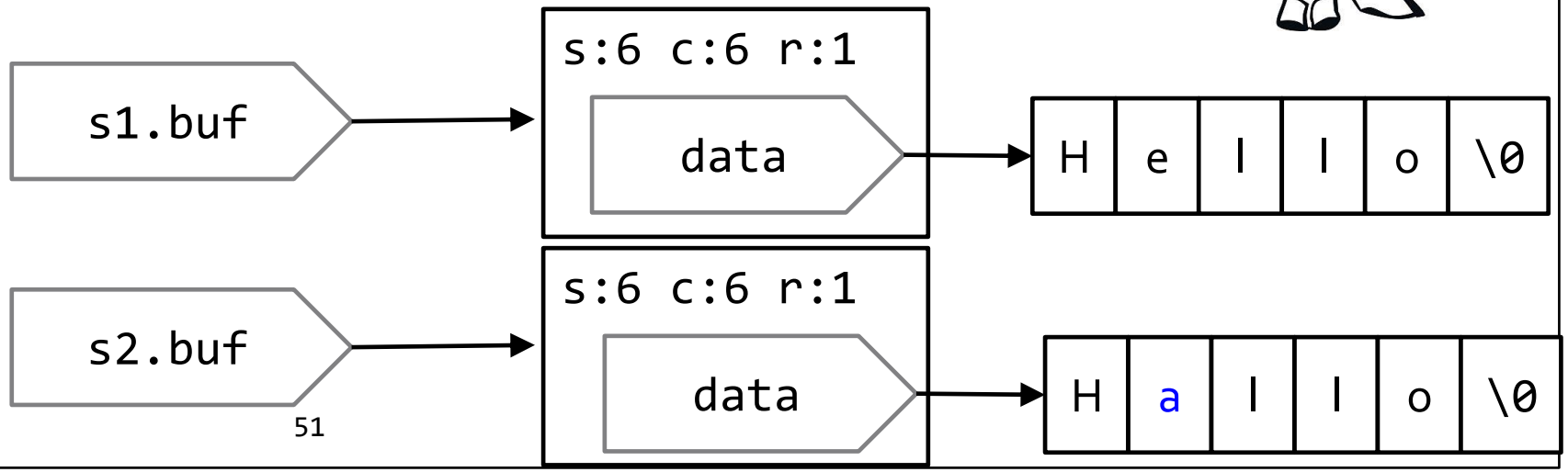
Copy On Write (идиома COW)



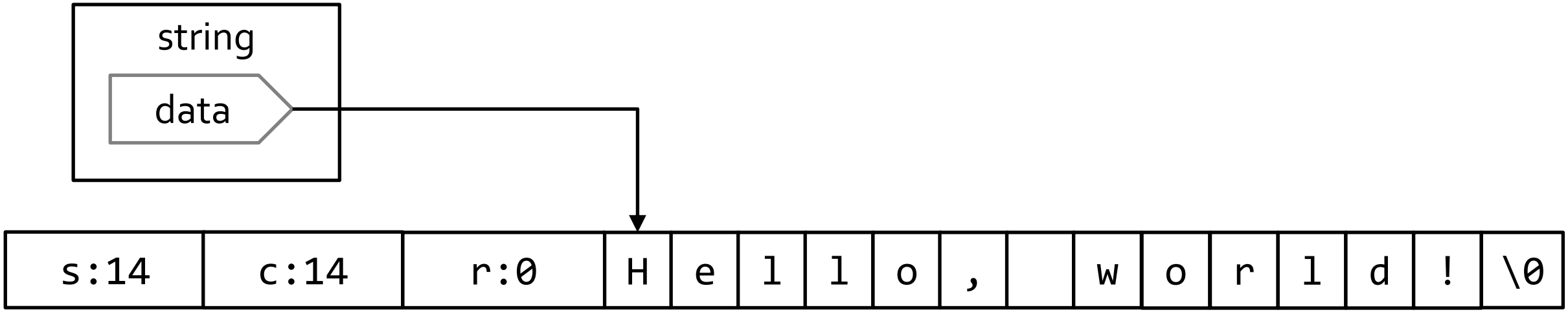
- Что если попробовать считать ссылки в строке?

```
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++



- Хранится счётчик ссылок -1, поэтому на рисунке он нулевой
- Активно используется COW

Обсуждение: COW

- С самого начала идиома имела своих сторонников и противников
- На какой стороне вы?

Обсуждение: COW

- С самого начала идиома имела своих сторонников и противников
- Экономия памяти
- Дешёвое копирование (просто инкремент счётчика ссылок)
- Меньше аллокаций и удалений в куче => прирост производительности
- Лишний уровень косвенности
- Вирусное проникновение копирования во все модифицирующие операции
- Проблемы thread safety (Multithread COW disease)
- Однако есть соображение, которое рушит баланс. Это инвалидация указателей

Инвалидация указателей

- Операции над строкой могут инвалидировать указатели внутрь строки. Например:

```
std::string a = "Hello";
const char *p = &a[3];
a += "world"; // после этой точки p нельзя использовать
```

- Здесь нет проблем.
- Проблема в том, что в случае COW указатели инвалидируются при совершенно безобидных операциях.

Инвалидация указателей

- Операции над строкой могут инвалидировать указатели внутрь строки. Например:

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

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

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

- Для non-COW строк p ещё валиден, но для COW может быть уже и нет.

Инвалидация указателей

- В 2011 году официально было запрещено инвалидировать указатели при выполнении `operator[] (C++11, 24.1.4.6)`.
- Это исключает COW-реализации `std::string`.
- Как желаемый итог: COW is (almost) dead.
- В реальности исключение из стандарта COW строк без введения достойной замены породило кучу велосипедов.

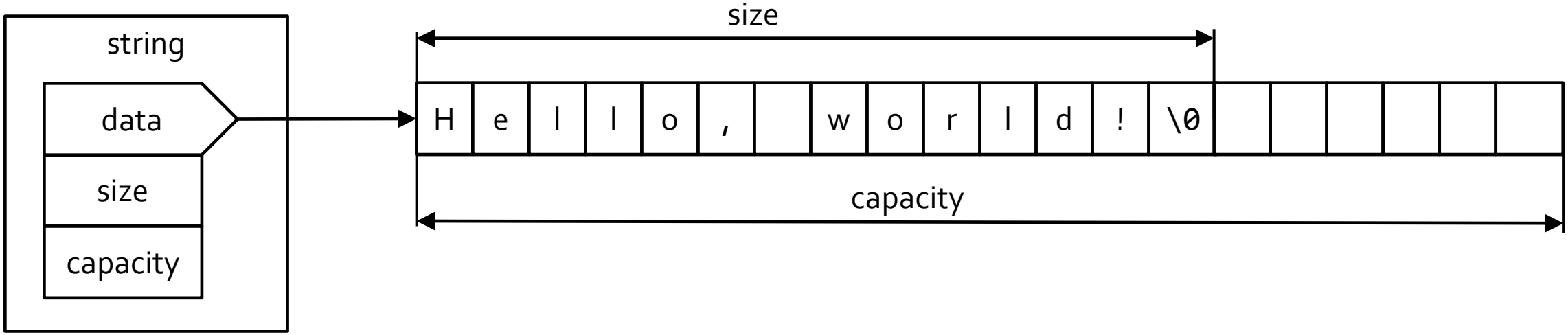
COW is (almost) dead

Нынешнее состояние

Желаемое состояние

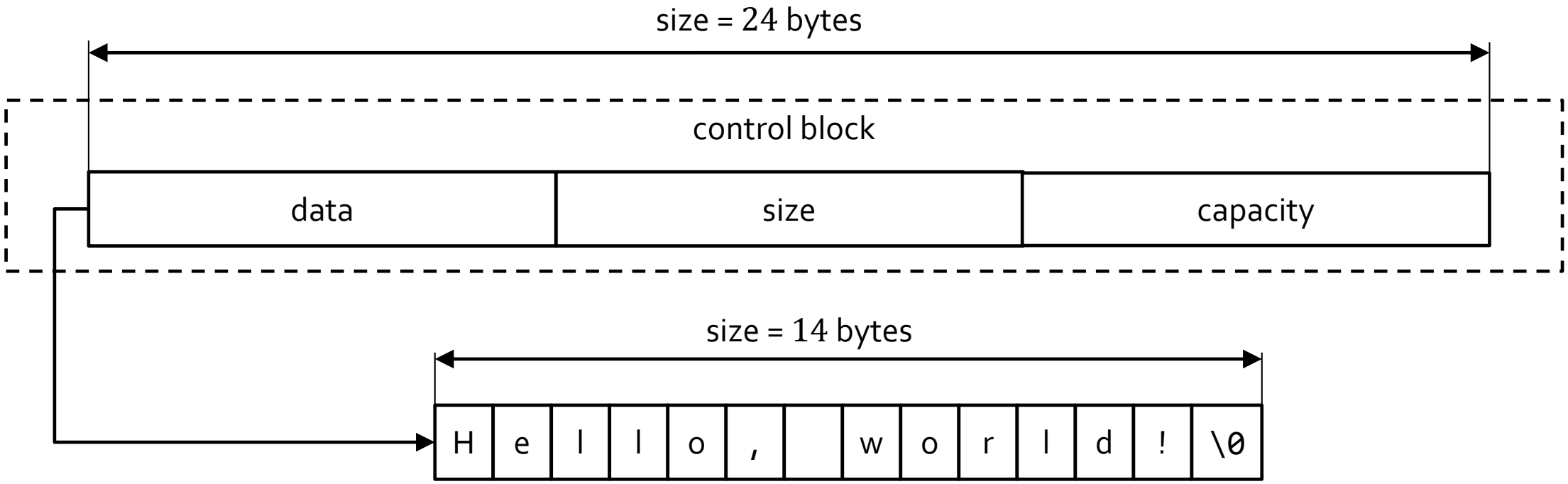
Но означает ли это, что мы совсем ничего не можем сделать на уровне проектирования класса?

Обсуждение: одна старая картинка



- Эта картинка врёт в одной очень существенной детали
- В какой?

Старая картинка, настоящий масштаб



- Небольшие данные вполне умещаются в control block.

Small string optimizations (SSO)

- Идея в целом была изложена на прошлом слайде: иногда настоящего выделения динамической памяти не нужно.

```
class string {
  size_type size_;
  union {
    struct {
      char *data_;
      size_type capacity_;
    } large_;
    char small_[sizeof(large_)];
  };
// ну и так далее
```

Обсуждение

- Какие минусы вы видите в таком подходе к SSO?

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

// ну и так далее
```

Обсуждение

- Какие минусы вы видите в таком подходе к SSO?
- Усложняется копирование.
- Становится нетривиальным перемещение.
- Добавляется время на выбор.

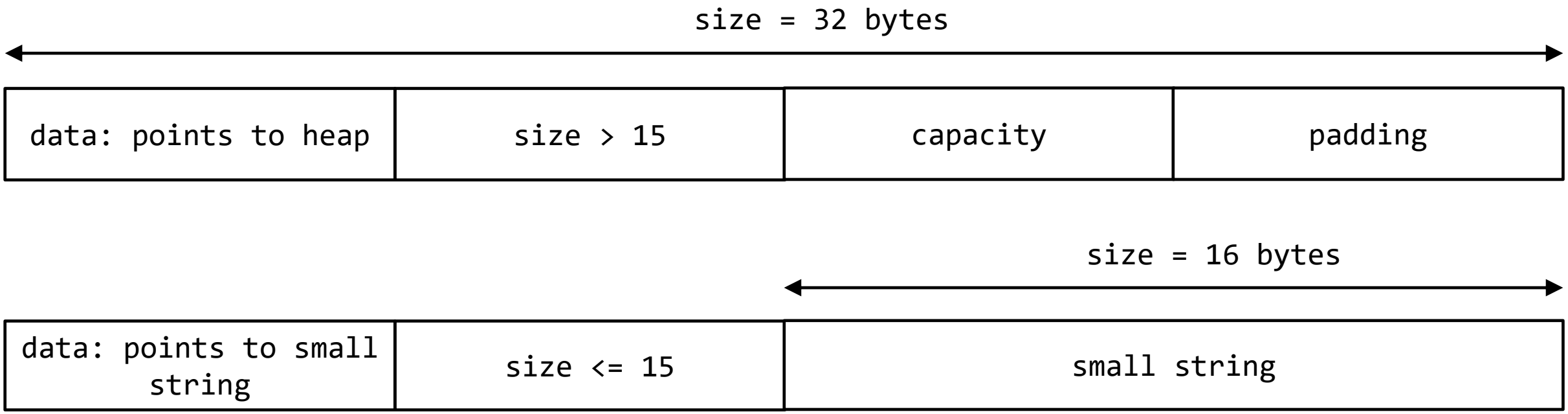
```
this->small_[i]
```

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

при каждом доступе (в том числе чтении) с проверкой размера.

- Последняя проблема серьёзней. Можно ли с этим что-нибудь сделать?

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



- Это решение позволяет избежать потерь времени при доступе
- Но уменьшает размер самой строки

Проблема: а теперь учтём UTF32

- В случае если один символ занимает не один байт (а, например, четыре) у SSO проблемы.
- Но в первую очередь проблемы у нас. Как обобщить разработанную строчку на символы разных размеров?
- Первая идея: написать три разных класса: `utf8string`, `utf16string` и `utf32string`.
- Покритикуйте эту идею.

Шаблон класса строки

- Как в принципе устроен basic_string.

```
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:
  // тут все его 89 методов
};
```

Определения для удобства

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

- Тут сознательно использован `typedef` а не `using`. Вы должны быть одинаковы хорошо знакомы с обоими способами определения синонимов.

Характеристики типов

- Есть много вопросов, ответы на которые разные для разных строк с разными типами символов
- Разумно свести всё это в класс

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

- Основные методы:
- `assign, eq, lt, move, compare, find, eof, ....`

```
template <typename CharT,
          typename Traits = std::char_traits<CharT>>
class basic_string {
// тут всё точно так же но с использованием Traits
```

Обсуждение

- Является ли способ выделения памяти на символ характеристикой символа?

Аллокаторы

- Выделение памяти абстрагирует аллокатор. Стандартный аллокатор сводится к чему-то вроде `malloc`.

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

class basic_string {

// тут всё точно так же но с использованием Traits и Allocator
```

Строки не только для символов

- Следующий код, к сожалению, не работает.

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

- Что будем делать?

https://godbolt.org/z/73MTKMr1o

Строки не только для символов

- Используем basic string!

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

- Не очень красиво, но класс basic_string_view не мутабельный.

Применяем span

```
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

Обсуждение

- Как вы разобьёте строку по сепараторам?

```
std::string str = ";;Hello|world||-foo--bar;yow;baz|";
```

- На экране (или в векторе строк) должно быть:

```
Hello world foo bar yow baz
```

Немного boost

- На слайде с функциональным соответствием был упомянут boost tokenizer.

```
std::string str = ";;Hello|world||-foo--bar;yow;baz|";

boost::char_separator<char> sep("-;|");

boost::tokenizer<char_separator<char>> tokens(str, sep);

for (auto tok : tokens)
  std::cout << "<" << tok << "> ";
```

- Основное внимание стоит обратить на параметризацию токенайзера разделителем и разделителя символьным типом.

https://godbolt.org/z/Mc95csoda

Обсуждение

- Не настало ли время теперь построить велосипед?

Задачи

- Найдите в стандарте правила использования println для char arrays. Обоснованы ли мои опасения?
- Прочитайте [conv.qual] и попробуйте построить алгоритм который берёт два типа (просто строчки, состоящие только из const, char, [] и *) проверяет сработает ли приведение.

```
testqual("const char**", "char**"); // -> false
```

- Напишите для std::basic_string сравнение (`operator==`).
  - Сделаете ли вы этот оператор методом класса или свободной функцией?
  - Как должны сравниваться строки с одинаковым `CharT`, но разными `Traits`?
  - А если у них одинаковые `CharT` и `Traits`, но разные аллокаторы?

Задачи (продолжение)

- Напишите собственный класс COW-строки.
  - Реализуйте его big-5.
  - Поддержите токенизацию.
  - Напишите метод, который ищет подстроку в строке.
- Напишите класс `string_twine` для $O(\log(N))$ конкатенации `string_view`.

```
std::string_view sv = "Hello,", sv2 = "World!";
auto s = string_twine(sv, " ", sv2).str(); // -> string
```

Литература

- 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