12RAII

Contents

1 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.3)
project(lec12 CXX)
include(.cs211/cmake/CMakeLists.txt)

add_test_program(test_string_view
        test/test_string_view.cxx
        src/string_view.cxx
        src/Owned_string.cxx)

add_test_program(test_Owned_string
        test/test_Owned_string.cxx
        src/Owned_string.cxx
        src/string_view.cxx)

# vim: ft=cmake
```

2 src/Owned_string.hxx

```
#pragma once

#include "string_view.hxx"

// A struct for representing a string. Unlike a C string
// ('\0'-terminated char array), this struct records the length of the
// string in the `size_` field, so `data_` can contain '\0's.
//
// (We will also '\0'-terminate it so that it can be passed to C
// functions that expect that, but C++ doesn't need that.)
//
// For this struct to be valid, some conditions (invariants) need to
// hold:
//
//  1. `capacity_` is 0 if and only if `data_` is null.
//
//  2. If `capacity_` is non-zero then it contains the actual allocated
//  size of the array object pointed to by `data_.`
//
//  3. `size_ + 1 <= capacity_`
//
//  4. The first `size_` elements of `data_` are initialized, and
//  `data_[size_] == '\0'`.
//
// The underscores on the member variable names ("member variable" is
// C++-speak for "field") are a convention meaning that they are
// *private*, which means that client code of this header should never
// access them directly, but always via the functions below. This is to
// ensure that the invariants above are never broken. C++ understand
// privacy and can prevent the client from accessing members, but we
// aren't going to use that feature yet.
class Owned_string
{
public:
    /*
     * Constructors.
     *
     * These are used to initialize a new `Owned_string` object, possibly
     * from some arguments. Every `Owned_string` object that is constructed
     * will be destroyed automatically using the destructor `~Owned_string`
     * declared below.
     */
```

```
43
44      // Initializes `this` to the empty string.
45      explicit Owned_string(size_t capacity = 0);
46
47      // Initializes `this` to from a range.
48      Owned_string(const char* begin, const char* end);
49
50      // Initializes `this` to a copy of the given string_view (this covers
51      // C strings and ranges as well):
52      Owned_string(string_view);
53
54      // Copy constructor: initializes `this` to be a copy of the
55      // argument.
56      Owned_string(Owned_string const&);
57
58      // Move constructor: initializes `this` by stealing the argument's
59      // memory, leaving it valid but empty.
60      Owned_string(Owned_string&&) noexcept;
61
62      /*
63       * Destructor.
64       */
65
66      // C++ calls this automatically whenever a `Owned_string` object needs to
67      // be destroyed (such as when it goes out of scope).
68      ~Owned_string();
69
70      /*
71       * Assignment "operators".
72       */
73
74      // Assigns the contents of the argument to `this`. This may reuse
75      // `this`'s memory or reallocate.
76      Owned_string& operator=(Owned_string const&);
77
78      // Steals the argument's memory, assigning it to `this` and leaving
79      // the argument object valid but empty.
80      Owned_string& operator=(Owned_string&&) noexcept;
81
82      /*
83       * Non-lifecycle `Owned_string` operations.
84       */
85
86      // Returns whether `this` is the empty string.
87      bool empty() const;
```

```
88
89      // Returns the number of characters in `this`. This does not
90      // depend on '\0' termination, and internal '\0's are allowed.
91      std::size_t size() const;
92
93      // Returns the character of `this` at the given index.
94      //
95      // ERROR: If `index >= this->size()` then the behavior is
96      // undefined.
97      char operator[](std::size_t index) const;
98
99      // Returns a reference to the character of `this` at the given index.
100     //
101     // ERROR: If `index >= this->size()` then the behavior is
102     // undefined.
103     char& operator[](std::size_t index);
104
105     // Adds character `c` to the end of `this`. This may cause pointers
106     // returned by previous calls to `this->operator[]` or `this->c_str`
107     // to become invalidated.
108     void push_back(char c);
109
110     // Removes the last character of the string.
111     //
112     // ERROR: UB if `Owned_string(this)`.
113     void pop_back();
114
115     // Expands the capacity, if necessary, to hold `additional_cap` more
116     // characters.
117     void reserve(std::size_t additional_cap);
118
119     // Empties the string in constant time. (Preserves any allocation.)
120     void clear();
121
122     // Swaps the contents of two strings without copying the actual
123     // characters.
124     void swap(Owned_string&);
125
126     // Appends another string onto this string.
127     Owned_string& operator+=(string_view that);
128
129     // Automatic conversion from `Owned_string` to `string_view`:
130     operator string_view() const;
131
132     // Returns a pointer to the content of this string as a C-style string.
```

```
133     // Note that internal '\0's will make `strlen(String_c_str(s))` less than
134     // `String_size(s)`, which doesn't depend on the content of the string.
135     const char* c_str() const;
136 
137     /*
138      * Some operations that were hard to define in lec09-String.h-style.
139      */
140 
141     // Returns a pointer to the first character of the string.
142     const char* begin() const;
143     char* begin();
144 
145     // Returns a pointer to the first character past the end of the
146     // string.
147     const char* end() const;
148     char* end();
149 
150     // Gives up ownership of the data pointer, leaving this string empty.
151     // The caller becomes the owner of the data pointer.
152     char* release() noexcept;
153 
154     // Takes ownership of `data`, which must have a size of `size` and
155     // capacity of at least `size + 1.
156     static Owned_string from_owned_parts(char* data, std::size_t size) noexcept;
157 
158     /*
159      * PRIVATE HELPER FUNCTIONS.
160      *
161      * These are functions that are useful for implementing the
162      * functions above. They should not be called by clients.
163      * Thus, they are documented in the implementation file, not
164      * in this header.
165      */
166 
167 private:
168     friend Owned_string operator+(string_view, string_view);
169 
170     // Private constructor used to set all the fields
171     Owned_string(size_t size, size_t capacity, const char* data);
172 
173     void set_empty_() noexcept;
174     void ensure_capacity_(std::size_t min_cap);
175     void prereserved_append_(string_view sv);
176 
177     /*
```

```
178         * Data members -- clients can't touch these.
179         *
180         * These are where the data is actually stored---everything above
181         * here declares operations (member functions), and below is data.
182         */
183
184     std::size_t size_;
185     std::size_t capacity_;
186     char*       data_;
187 };
188
189 /*
190  * These are free functions. Some of them *could* be defined as members,
191  * but it improves encapsulation to minimize the number of members of
192  * possible.
193  */
194
195 // Appends two strings, returning a new string.
196 Owned_string operator+(string_view, string_view);
197
198 // Steals the left string's memory to append the right and return the
199 // result.
200 Owned_string operator+(Owned_string&&, string_view);
```

3 src/Owned_string.cxx

```
1  #include "Owned_string.hxx"
2  #include "string_view.hxx"
3
4  #include <algorithm>
5  #include <cstdlib>
6  #include <cstring>
7
8  static size_t cap_for(size_t size)
9  {
10     return size? size + 1 : size;
11 }
12
13 // Constructs an empty string of the given capacity. This serves
14 // and the default constructor as well.
15 Owned_string::Owned_string(size_t capacity)
16         : size_(0)
17         , capacity_(capacity)
```

```
18          , data_(capacity? new char[capacity] : nullptr)
19  { }
20
21  // Private constructor delegates to allocating constructor above,
22  // then copies if there's anything to copy.
23  Owned_string::Owned_string(size_t size, size_t capacity, const char* data)
24          : Owned_string(capacity)
25  {
26      if (size)
27          prereserved_append_({data, size});
28  }
29
30  Owned_string::Owned_string(const char* begin, const char* end)
31          : Owned_string(string_view(begin, end))
32  { }
33
34  Owned_string::Owned_string(string_view sv)
35          : Owned_string(sv.size(), cap_for(sv.size()), sv.begin())
36  { }
37
38  Owned_string::Owned_string(Owned_string const& that)
39          : Owned_string(that.size(), cap_for(that.size()), that.begin())
40  { }
41
42  Owned_string::Owned_string(Owned_string&& that) noexcept
43          : size_(that.size_)
44          , capacity_(that.capacity_)
45          , data_(that.data_)
46  {
47      that.set_empty_();
48  }
49
50  /*
51   * Destructor.
52   */
53
54  Owned_string::~Owned_string()
55  {
56      delete [] data_;
57  }
58
59  /*
60   * Assignment "operators".
61   */
62
```

```
63  Owned_string& Owned_string::operator=(Owned_string const& that)
64  {
65      if (this != &that) {
66          clear();
67          *this += that;
68      }
69
70      return *this;
71  }
72
73  Owned_string& Owned_string::operator=(Owned_string&& that) noexcept
74  {
75      Owned_string temp(std::move(that));
76      swap(temp);
77      return *this;
78  }
79
80  /*
81   * Other `String` operations.
82   */
83
84  bool Owned_string::empty() const
85  {
86      return size_ == 0;
87  }
88
89  size_t Owned_string::size() const
90  {
91      return size_;
92  }
93
94  char Owned_string::operator[](size_t index) const
95  {
96      return data_[index];
97  }
98
99  char& Owned_string::operator[](size_t index)
100 {
101     return data_[index];
102 }
103
104 void Owned_string::push_back(char c)
105 {
106     // Make sure we have room for at least one more character:
107     reserve(1);
```

```
108     data_[size_++] = c;
109     data_[size_] = '\0';
110 }
111
112 void Owned_string::pop_back()
113 {
114     data_[--size_] = '\0';
115 }
116
117 void Owned_string::reserve(size_t additional_cap)
118 {
119     ensure_capacity_(cap_for(size_ + additional_cap));
120 }
121
122 void Owned_string::clear()
123 {
124     if (data_) {
125         size_  = 0;
126         *end() = '\0';
127     }
128 }
129
130 void Owned_string::swap(Owned_string& that)
131 {
132     std::swap(size_,     that.size_);
133     std::swap(capacity_, that.capacity_);
134     std::swap(data_,     that.data_);
135 }
136
137 Owned_string& Owned_string::operator+=(string_view that)
138 {
139     reserve(that.size());
140     prereserved_append_(that);
141     return *this;
142 }
143
144 const char* Owned_string::c_str() const
145 {
146     return data_? data_ : "";
147 }
148
149 Owned_string::operator string_view() const
150 {
151     if (empty())
152         return {};
```

```
153     else
154         return {begin(), end()};
155 }
156 
157 const char* Owned_string::begin() const
158 {
159     return data_;
160 }
161 
162 char* Owned_string::begin()
163 {
164     return data_;
165 }
166 
167 const char* Owned_string::end() const
168 {
169     return data_ + size_;
170 }
171 
172 char* Owned_string::end()
173 {
174     return data_ + size_;
175 }
176 
177 char* Owned_string::release() noexcept
178 {
179     char* result = data_;
180     set_empty_();
181     return result;
182 }
183 
184 // Clears this `String` to be the empty string with no allocated memory.
185 // Note that if `data_` points to a live object then calling this
186 // function could leak it.
187 void Owned_string::set_empty_() noexcept
188 {
189     size_ = capacity_ = 0;
190     data_ = nullptr;
191 }
192 
193 // Makes sure that this string has at least capacity `min_cap`, growing
194 // it if necessary.
195 void Owned_string::ensure_capacity_(size_t min_cap)
196 {
197     if (capacity_ < min_cap) {
```

```
198         Owned_string temp(size_,
199                           std::max(min_cap, 2 * capacity_),
200                           data_);
201         swap(temp);
202     }
203 }
204
205 void Owned_string::prereserved_append_(string_view sv)
206 {
207     std::copy(sv.begin(), sv.end(), end());
208     size_ += sv.size();
209     *end() = '\0';
210 }
211
212 Owned_string Owned_string::from_owned_parts(char* data, std::size_t size) noexcept
213 {
214     Owned_string result;
215
216     if (data) {
217         result.size_     = size;
218         result.capacity_ = cap_for(size);
219         result.data_     = data;
220     }
221
222     return result;
223 }
224
225 Owned_string operator+(string_view a, string_view b)
226 {
227     Owned_string result;
228     result.reserve(a.size() + b.size());
229     result.prereserved_append_(a);
230     result.prereserved_append_(b);
231     return result;
232 }
233
234 Owned_string operator+(Owned_string&& a, string_view b)
235 {
236     a += b;
237     return std::move(a);
238 }
```

4 src/string_view.hxx

```
#pragma once

#include <iostream>

// Class for representing ranges of characters, usually for the
// purpose of comparing them. The idea is that we have constructors for
// a bunch of different ways that strings might be specified, and each
// gets converted to a string_view [begin, end).
//
// Enforces invariant that begin <= end; throws otherwise.
class string_view
{
public:
    // Constructs an empty string_view.
    string_view();

    // Constructs a string_view from `begin` and `end` directly.
    string_view(const char* begin, const char* end);

    // Constructs a string_view from the start and the size.
    string_view(const char*, size_t);

    // Constructs a string_view from a '\0'-terminated C-style string.
    template <class T>
    string_view(T const* const&);

    // Constructs a string_view from a string literal using its static size.
    template <class T, size_t N>
    string_view(T const (&s) [N]);

    /*
     * Getters
     */

    // Returns the number of characters.
    size_t size() const;

    // Returns whether this string_view is empty.
    bool empty() const;

    // Gets the pointer to the first character.
    const char* begin() const;
```

```
43
44      // Gets the pointer one past the last character.
45      const char* end() const;
46
47      // Indexes.
48      //
49      // PRECONDITION (unchecked): index < size()
50      char operator[](size_t index) const;
51
52      // Returns the substring from index `start` to the end.
53      //
54      // PRECONDITION (asserted):
55      //  - start is in bounds
56      string_view substring(size_t start) const;
57
58      // Returns the substring from index `start` of length `size`.
59      //
60      // PRECONDITION (asserted):
61      //  - [start, start + size) is in bounds
62      string_view substring(size_t start, size_t size) const;
63
64  private:
65      /*
66       * Member variables
67       */
68
69      const char* begin_;
70      const char* end_;
71      // INVARIANT: begin_ <= end_
72  };
73
74  // Overloads == for `string_view`s.
75  bool operator==(string_view, string_view);
76
77  // Overloads != for `string_view`s.
78  bool operator!=(string_view, string_view);
79
80  template <class T>
81  string_view::string_view(T const* const& s)
82          : string_view(s, std::strlen(s))
83  { }
84
85  // Templates need to be defined in the .h file, not the .cpp file.
86  // (We subtract 1 from N because N will include the string literal's
87  // '\0' terminator.)
```

```
88  template <class T, size_t N>
89  string_view::string_view(T const (&s) [N])
90          : string_view(s, N - 1)
91  { }
92
93  // Overloads stream insertion (printing)
94  std::ostream& operator<<(std::ostream&, string_view);
```

5 src/string_view.cxx

```
1   #include "string_view.hxx"
2   #include <algorithm>
3   #include <cassert>
4   #include <cstring>
5
6   static char const* const empty_string = "";
7
8   string_view::string_view()
9           : begin_(empty_string)
10          , end_(empty_string)
11  { }
12
13  string_view::string_view(const char* begin, const char* end)
14      : begin_(begin)
15      , end_(end)
16  {
17      if (begin == nullptr)
18          throw std::invalid_argument("string_view: begin == nullptr");
19
20      if (end == nullptr)
21          throw std::invalid_argument("string_view: end == nullptr");
22
23      if (begin > end)
24          throw std::invalid_argument("string_view: begin > end");
25  }
26
27  string_view::string_view(const char* s, size_t size)
28      : string_view(s, s + size)
29  { }
30
31  size_t string_view::size() const
32  {
33      return end_ - begin_;
```

```
34  }
35
36  bool string_view::empty() const
37  {
38      return end_ == begin_;
39  }
40
41  const char* string_view::begin() const
42  {
43      return begin_;
44  }
45
46  const char* string_view::end() const
47  {
48      return end_;
49  }
50
51  char string_view::operator[](size_t i) const
52  {
53      return begin_[i];
54  }
55
56  string_view string_view::substring(size_t start) const
57  {
58      assert( start <= end_ - begin_ );
59      return {begin_ + start, end_};
60  }
61
62  string_view string_view::substring(size_t start, size_t size) const
63  {
64      assert( start <= SIZE_MAX - size );
65      assert( start + size <= end_ - begin_ );
66      return {begin_ + start, begin_ + start + size};
67  }
68
69  bool operator==(string_view sv1, string_view sv2)
70  {
71      return sv1.size() == sv2.size() &&
72          std::equal(sv1.begin(), sv1.end(), sv2.begin());
73  }
74
75  bool operator!=(string_view sv1, string_view sv2)
76  {
77      return !(sv1 == sv2);
78  }
```

```
79
80  std::ostream& operator<<(std::ostream& os, string_view sv)
81  {
82      return os.write(sv.begin(), sv.size());
83  }
```

6 test/test_Owned_string.cxx

```
1   #include "Owned_string.hxx"
2   #include "string_view.hxx"
3
4   #include <catch.hxx>
5
6   #include <algorithm>
7   #include <cstring>
8
9   // Tests that default construction works as expected.
10  TEST_CASE("default construction")
11  {
12      Owned_string s;
13      CHECK(s.empty());
14      CHECK(s.size() == 0);
15      CHECK(s == "");
16  }
17
18  // Tests the c_str constructor on the given C string `cs`.
19  static void c_str_constructor_case(const char* cs)
20  {
21      Owned_string s(cs);
22      CHECK(s.empty() == (cs[0] == '\0'));
23      CHECK(s.size() == std::strlen(cs));
24      CHECK(s == cs);
25  }
26
27  // Test cases for `Owned_string_construct_c_str`.
28  TEST_CASE("c string constructor")
29  {
30      c_str_constructor_case("");
31      c_str_constructor_case("hello world");
32      c_str_constructor_case("hello\0world");
33  }
34
35  // Tests the range constructor on the given range [cp, cp + len).
```

```
36  static void range_constructor_case(string_view r)
37  {
38      Owned_string s(r);
39      CHECK(s.empty() == (r.size() == 0));
40      CHECK(s.size() == r.size());
41      CHECK(s == r);
42  }
43
44  // Test cases for `Owned_string_construct_range`.
45  TEST_CASE("range constructor")
46  {
47      range_constructor_case("");
48      range_constructor_case("hello world");
49      range_constructor_case("hello\0world");
50      range_constructor_case("hello\0world");
51      range_constructor_case("hello\0world");
52  }
53
54  // Tests the copy constructor by copying a string that is first
55  // constructed from the range [cp, cp + len).
56  static void copy_constructor_case(string_view r)
57  {
58      Owned_string s1(r.begin(), r.end());
59      Owned_string s2(s1);
60
61      CHECK(s1 == r);
62      CHECK(s2 == r);
63      CHECK(s1 == s2);
64
65      if (r.size() > 0) CHECK(s1.c_str() != s2.c_str());
66  }
67
68  // Tests the copy constructor.
69  TEST_CASE("copy constructor")
70  {
71      copy_constructor_case("");
72      copy_constructor_case("hello");
73      copy_constructor_case("hello\0world");
74  }
75
76  // Tests the move constructor by moving a string that is first
77  // constructed from the range [cp, cp + len).
78  static void move_constructor_case(string_view r)
79  {
80      Owned_string s1(r.begin(), r.end());
```

```
81      CHECK(s1 == r);
82
83      Owned_string s2(std::move(s1));
84      CHECK(s1 == "");
85      CHECK(s2 == r);
86  }
87
88  // Tests the move constructor.
89  TEST_CASE("move constructor")
90  {
91      move_constructor_case("");
92      move_constructor_case("hello");
93      move_constructor_case("hello\0world");
94  }
95
96  // Tests the copy-assignment operator by constructing strings
97  // from the ranges [cs1, cs1 + len1) and [cs2, cs2 + len2), and
98  // then copy-assigning the latter to the former.
99  static void copy_assignment_case(string_view r1, string_view r2)
100 {
101     Owned_string s1(r1.begin(), r1.end());
102     Owned_string s2(r2.begin(), r2.end());
103
104     const char* old1 = s1.begin();
105
106     s1 = s2;
107     CHECK(s1 == s2);
108
109     const char* new1 = s1.begin();
110
111     if (r1.size() >= r2.size()) {
112         CHECK(new1 == old1);
113     }
114 }
115
116 // Tests Owned_string_assign_copy.
117 TEST_CASE("copy assignment")
118 {
119     copy_assignment_case("", "hello");
120     copy_assignment_case("hello", "");
121     copy_assignment_case("howdy", "hello world");
122     copy_assignment_case("hello world", "howdy");
123     copy_assignment_case("howdy", "hello\0world");
124 }
125
```

```
126 // Tests the move-assignment operator by constructing strings
127 // from the ranges [cs1, cs1 + len1) and [cs2, cs2 + len2), and
128 // then move-assigning the latter to the former.
129 static void move_assignment_case(string_view r1, string_view r2)
130 {
131     Owned_string s1(r1.begin(), r1.end());
132     Owned_string s2(r2.begin(), r2.end());
133 
134     const char* old2 = s2.c_str();
135     s1 = std::move(s2);
136     CHECK(s2.empty());
137     CHECK(s1 == r2);
138     const char* new1 = s1.c_str();
139 
140     // Moving means that the pointer owned by `s1` now is the same
141     // as the pointer owned by `s2` before.
142     CHECK(new1 == old2);
143 }
144 
145 // Tests Owned_string_assign_move.
146 TEST_CASE("move assignment")
147 {
148     move_assignment_case("", "hello");
149     move_assignment_case("hello", "");
150     move_assignment_case("howdy", "hello world");
151     move_assignment_case("hello world", "howdy");
152     move_assignment_case("howdy", "hello\0world");
153 }
154 
155 // Tests push_back by first constructing a string from the
156 // range [cp1, cp1 + len1), and then pushing back each character
157 // in the range [cp2, cp2 + len2) in turn.
158 static void push_back_case(string_view r1, string_view r2)
159 {
160     size_t size1 = r1.size();
161     size_t size2 = r2.size();
162 
163     char* buf = new char[size1 + size2];
164     std::copy(r1.begin(), r1.end(), buf);
165     std::copy(r2.begin(), r2.end(), buf + size1);
166 
167     Owned_string s(r1.begin(), r1.end());
168 
169     for (size_t i = 0; i < size2; ++i) {
170         s.push_back(r2[i]);
```

```
171         CHECK(s.size() == size1 + i + 1);
172         CHECK(s == string_view(buf, size1 + i + 1));
173     }
174 
175     delete [] buf;
176 }
177 
178 // Tests Owned_string_push_back.
179 TEST_CASE("push_back")
180 {
181     push_back_case("", "");
182     push_back_case("", "hello");
183     push_back_case("", "hello\0world");
184     push_back_case("C++", "");
185     push_back_case("C++", "hello");
186     push_back_case("C++", "hello\0world");
187     push_back_case("hello\0C++\n", "");
188     push_back_case("hello\0C++\n", "hello");
189     push_back_case("hello\0C++\n", "hello\0world");
190 }
191 
192 // Tests pop_back by first constructing a string from the
193 // range [cp1, cp1 + len1), and then popping the last character
194 // repeatedly until it's empty.
195 static void pop_back_case(string_view r)
196 {
197     size_t const size = r.size();
198     Owned_string s(r);
199 
200     for (size_t i = 0; i < size; ++i) {
201         s.pop_back();
202         CHECK(s.size() == size - i - 1);
203         CHECK(s == string_view(r.begin(), size - i - 1));
204     }
205 }
206 
207 // Tests `Owned_string_pop_back`.
208 TEST_CASE("pop_back")
209 {
210     pop_back_case("");
211     pop_back_case("");
212     pop_back_case("");
213     pop_back_case("C++");
214     pop_back_case("C++");
215     pop_back_case("C++");
```

```
216     pop_back_case("hello\0C++\n");
217     pop_back_case("hello\0C++\n");
218     pop_back_case("hello\0C++\n");
219 }
220 
221 // Tests `Owned_string_index` and `Owned_string_index_mut`.
222 TEST_CASE("operator[]")
223 {
224     Owned_string s1("hello, world");
225     Owned_string const& s2 = s1;
226     CHECK(s2[0] == 'h');
227     CHECK(s2[1] == 'e');
228     CHECK(s2[2] == 'l');
229     s1[0] = 'H';
230     s1[6] = '\0';
231     CHECK(s2 == "Hello,\0world");
232 }
233 
234 TEST_CASE("operator+")
235 {
236     const char* bar = "b\0r";
237 
238     Owned_string s("foo");
239     CHECK(s == "foo");
240     s += s;
241     CHECK(s == "foofoo");
242     s = s + s;
243     CHECK(s == "foofoofoofoo");
244     s = Owned_string(bar, bar + 3) + "foo";
245     CHECK(s == "b\0rfoo");
246 }
```

7 test/test_string_view.cxx

```
1 #include "string_view.hxx"
2 #include "Owned_string.hxx"
3 #include <catch.hxx>
4 #include <sstream>
5 
6 static const char hello[]       = "hello";
7 static const char hello0world[] = "hello\0world";
8 
9 TEST_CASE("invariant")
```

```
10  {
11      string_view(hello, hello + 3);
12      CHECK_THROWS_AS(string_view(hello + 3, hello), std::invalid_argument);
13  }
14
15  TEST_CASE("stream insertion (printing)")
16  {
17      string_view sv1("hello");
18      string_view sv2(hello0world, 11);
19
20      std::ostringstream oss;
21      oss << sv1;
22      CHECK( oss.str() == "hello" );
23
24      oss.str("");
25      oss << sv2;
26      CHECK( oss.str() == std::string(hello0world, 11) );
27  }
28
29  TEST_CASE("construction")
30  {
31      Owned_string s3(hello0world);
32
33      // Constructing from [begin, end) range:
34
35      string_view r11(hello, hello + 5);
36      CHECK(r11.size() == 5);
37
38      string_view r12(std::begin(hello0world), std::end(hello0world) - 1);
39      CHECK(r12.size() == 11);
40
41      string_view r13(s3);
42      CHECK(r13.size() == 11);
43
44      // Constructing from start, length:
45
46      string_view r21(hello, 5);
47      CHECK(r21.size() == 5);
48
49      string_view r22(hello0world, sizeof hello0world - 1);
50      CHECK(r22.size() == 11);
51
52      string_view r23(s3.c_str(), s3.size());
53      CHECK(r23.size() == 11);
54
```

```
55      // Constructing from sized array or `String`:
56
57      string_view r31 = hello;
58      CHECK(r31.size() == 5);
59
60      string_view r32 = hello0world;
61      CHECK(r32.size() == 11);
62
63      string_view r33 = s3;
64      CHECK(r33.size() == 11);
65  }
```