11ACCESS

Contents

1 CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.3)
2  project(lec11 CXX)
3  include(.cs211/cmake/CMakeLists.txt)
4
5  add_test_program(test_Owned_string
6          test/test_Owned_string.cxx
7          src/Owned_string.cxx
8          src/string_view1.cxx)
9
10 add_test_program(test_string_view1
11         test/test_string_view1.cxx
12         src/string_view1.cxx
13         src/Owned_string.cxx)
14
```

```
add_test_program(test_string_view2
        test/test_string_view2.cxx
        src/string_view2.cxx
        src/Owned_string.cxx)
```

2 src/Owned_string.hxx

```
#pragma once

#include <cstddef>

// A class for representing a string. Unlike a C string
// ('\0'-terminated char array), this class 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 an instance of this class 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.
```

```
37          *
38          * These are used to initialize a new `Owned_string` object, possibly
39          * from some arguments. Every `Owned_string` object that is constructed
40          * will be destroyed automatically using the destructor `~Owned_string`
41          * declared below.
42          */
43
44      // Initializes `this` to the empty string.
45      Owned_string();
46
47      // Initializes `this` to a copy of the given C ('\0'-terminated)
48      // string.
49      Owned_string(const char*);
50
51      // Initializes `this` to a copy of the array starting at `begin` and
52      // ending at (not including) `end`. This array may contain '\0's, and
53      // they will be copied too.
54      Owned_string(const char* begin, const char* end);
55
56      // Copy constructor: initializes `this` to be a copy of the
57      // argument.
58      Owned_string(Owned_string const&);
59
60      // Move constructor: initializes `this` by stealing the argument's
61      // memory, leaving it valid but empty.
62      Owned_string(Owned_string&&) noexcept;
63
64      /*
65       * Destructor.
66       */
67
68      // C++ calls this automatically whenever a `Owned_string` object needs to
69      // be destroyed (such as when it goes out of scope).
70      ~Owned_string();
71
72      /*
73       * Assignment "operators".
74       */
75
76      // Assigns the contents of the argument to `this`. This may reuse
77      // `this`'s memory or reallocate.
78      Owned_string& operator=(Owned_string const&);
79
80      // Steals the argument's memory, assigning it to `this` and leaving
81      // the argument object valid but empty.
```

```
82      Owned_string& operator=(Owned_string&&) noexcept;
83
84      /*
85       * Non-lifecycle `Owned_string` operations.
86       */
87
88      // Returns whether `this` is the empty string.
89      bool empty() const;
90
91      // Returns the number of characters in `this`. This does not
92      // depend on '\0' termination, and internal '\0's are allowed.
93      size_t size() const;
94
95      // Returns the character of `this` at the given index.
96      //
97      // ERROR: If `index >= this->size()` then the behavior is
98      // undefined.
99      char operator[](size_t index) const;
100
101     // Returns a reference to the character of `this` at the given index.
102     //
103     // ERROR: If `index >= this->size()` then the behavior is
104     // undefined.
105     char& operator[](size_t index);
106
107     // Adds character `c` to the end of `this`. This may cause pointers
108     // returned by previous calls to `this->operator[]` or `this->c_str`
109     // to become invalidated.
110     void push_back(char c);
111
112     // Removes the last character of the string.
113     //
114     // ERROR: UB if `this->empty()`.
115     void pop_back();
116
117     // Expands the capacity, if necessary, to hold `additional_cap` more
118     // characters.
119     void reserve(size_t additional_cap);
120
121     // Swaps the contents of two strings without copying the actual
122     // characters.
123     void swap(Owned_string&);
124
125     // Appends another string onto this string.
126     Owned_string& operator+=(Owned_string const& that);
```

```
127
128     // Appends a C string onto this string.
129     Owned_string& operator+=(const char* that);
130
131     // Returns a pointer to the content of this string as a C-style string.
132     // Note that internal '\0's will make `strlen(String_c_str(s))` less than
133     // `String_size(s)`, which doesn't depend on the content of the string.
134     const char* c_str() const;
135
136     /*
137      * Some operations that were hard to define in lec09-String.h-style.
138      */
139
140     // Returns a pointer to the first character of the string.
141     const char* begin() const;
142     char* begin();
143
144     // Returns a pointer to the first character past the end of the
145     // string.
146     const char* end() const;
147     char* end();
148
149     /*
150      * PRIVATE HELPER FUNCTIONS.
151      *
152      * These are functions that are useful for implementing the
153      * functions above. They should not be called by clients.
154      * Thus, they are documented in the implementation file, not
155      * in this header.
156      */
157
158 private:
159     void set_empty_();
160     void ensure_capacity_(size_t min_cap);
161     void append_range_(const char* begin, const char* end);
162
163     /*
164      * Data members -- clients can't touch these.
165      *
166      * These are where the data is actually stored---everything above
167      * here declares operations (member functions), and below is data.
168      */
169
170     size_t size_;
171     size_t capacity_;
```

```
172     char*  data_;
173 };
174 
175 /*
176  * These are free functions. Some of them *could* be defined as members,
177  * but it improves encapsulation to minimize the number of members of
178  * possible.
179  */
180 
181 // Appends two strings, returning a new string.
182 Owned_string operator+(Owned_string const&, Owned_string const&);
183 
184 // Appends two strings, returning a new string.
185 Owned_string operator+(Owned_string const&, const char*);
186 
187 // Appends two strings, returning a new string.
188 Owned_string operator+(const char*, Owned_string const&);
189 
190 // Steals the left string's memory to append the right and return the
191 // result.
192 Owned_string operator+(Owned_string&&, Owned_string const&);
193 
194 // Steals the left string's memory to append the right and return the
195 // result.
196 Owned_string operator+(Owned_string&&, const char*);
```

3 src/Owned_string.cxx

```
1  #include "Owned_string.hxx"
2  
3  #include <algorithm>
4  #include <cstdlib>
5  #include <cstring>
6  
7  Owned_string::Owned_string()
8          : size_{0}
9          , capacity_{0}
10         , data_{nullptr}
11 { }
12 
13 Owned_string::Owned_string(const char* s)
14     : Owned_string(s, s + strlen(s))
15 { }
```

```
16
17  Owned_string::Owned_string(const char* begin, const char* end)
18      : size_(begin <= end ? end - begin : 0)
19      , capacity_{size_ ? size_ + 1 : 0}
20      , data_{capacity_ ? new char[capacity_] : nullptr}
21  {
22      if (data_) {
23          std::copy(begin, end, data_);
24          data_[size_] = '\0';
25      }
26  }
27
28  Owned_string::Owned_string(Owned_string const& that)
29          : Owned_string(that.begin(), that.end())
30  { }
31
32  Owned_string::Owned_string(Owned_string&& that) noexcept
33          : size_{that.size_}
34          , capacity_{that.capacity_}
35          , data_{that.data_}
36  {
37      that.set_empty_();
38  }
39
40  /*
41   * Destructor.
42   */
43
44  Owned_string::~Owned_string()
45  {
46      delete [] data_;
47  }
48
49  /*
50   * Assignment "operators".
51   */
52
53  Owned_string& Owned_string::operator=(Owned_string const& that)
54  {
55      if (this == &that) return *this;
56
57      if (that.size_ > 0 && that.size_ + 1 >= capacity_) {
58          char* new_data  = new char[that.size_ + 1];
59          delete [] data_;
60          data_     = new_data;
```

```
61          capacity_ = that.size_ + 1;
62      }
63
64      if (data_) {
65          if (that.data_)
66              std::copy(that.begin(), that.end() + 1, data_);
67          else
68              data_[0] = '\0';
69      }
70
71      size_ = that.size_;
72
73      return *this;
74  }
75
76  Owned_string& Owned_string::operator=(Owned_string&& that) noexcept
77  {
78      Owned_string temp(std::move(that));
79      swap(temp);
80      return *this;
81  }
82
83  /*
84   * Other `String` operations.
85   */
86
87  bool Owned_string::empty() const
88  {
89      return size_ == 0;
90  }
91
92  size_t Owned_string::size() const
93  {
94      return size_;
95  }
96
97  char Owned_string::operator[](size_t index) const
98  {
99      return data_[index];
100 }
101
102 char& Owned_string::operator[](size_t index)
103 {
104     return data_[index];
105 }
```

```
106
107 void Owned_string::push_back(char c)
108 {
109     // Make sure we have room for at least one more character:
110     reserve(1);
111     data_[size_++] = c;
112     data_[size_] = '\0';
113 }
114
115 void Owned_string::pop_back()
116 {
117     data_[--size_] = '\0';
118 }
119
120 void Owned_string::reserve(size_t additional_cap)
121 {
122     if (size_ || additional_cap)
123         ensure_capacity_(size_ + additional_cap + 1);
124 }
125
126 void Owned_string::swap(Owned_string& that)
127 {
128     std::swap(size_, that.size_);
129     std::swap(capacity_, that.capacity_);
130     std::swap(data_, that.data_);
131 }
132
133 Owned_string& Owned_string::operator+=(Owned_string const& that)
134 {
135     append_range_(that.begin(), that.end());
136     return *this;
137 }
138
139 Owned_string& Owned_string::operator+=(const char* that)
140 {
141     append_range_(that, that + strlen(that));
142     return *this;
143 }
144
145 const char* Owned_string::c_str() const
146 {
147     return data_? data_ : "";
148 }
149
150 Owned_string operator+(Owned_string const& a, Owned_string const& b)
```

```
151 {
152     Owned_string result;
153     result.reserve(a.size() + b.size());
154     result += a;
155     result += b;
156     return result;
157 }
158 
159 Owned_string operator+(Owned_string const& a, const char* b)
160 {
161     Owned_string result;
162     result.reserve(a.size() + std::strlen(b));
163     result += a;
164     result += b;
165     return result;
166 }
167 
168 Owned_string operator+(const char* a, Owned_string const& b)
169 {
170     Owned_string result;
171     result.reserve(std::strlen(a) + b.size());
172     result += a;
173     result += b;
174     return result;
175 }
176 
177 Owned_string operator+(Owned_string&& a, Owned_string const& b)
178 {
179     a += b;
180     return std::move(a);
181 }
182 
183 Owned_string operator+(Owned_string&& a, const char* b)
184 {
185     a += b;
186     return std::move(a);
187 }
188 
189 const char* Owned_string::begin() const
190 {
191     return data_;
192 }
193 
194 char* Owned_string::begin()
195 {
```

```
196     return data_;
197 }
198 
199 const char* Owned_string::end() const
200 {
201     return data_ + size_;
202 }
203 
204 char* Owned_string::end()
205 {
206     return data_ + size_;
207 }
208 
209 // Clears this `String` to be the empty string with no allocated memory.
210 // Note that if `data_` points to a live object then calling this
211 // function could leak it.
212 void Owned_string::set_empty_()
213 {
214     size_ = capacity_ = 0;
215     data_ = nullptr;
216 }
217 
218 // Makes sure that this string has at least capacity `min_cap`, growing
219 // it if necessary.
220 void Owned_string::ensure_capacity_(size_t min_cap)
221 {
222     if (capacity_ < min_cap) {
223         size_t new_cap = std::max(min_cap, 2 * capacity_);
224         char* new_data = new char[new_cap];
225         if (data_)
226             std::copy(begin(), end() + 1, new_data);
227         delete [] data_;
228         data_ = new_data;
229         capacity_ = new_cap;
230     }
231 }
232 
233 void Owned_string::append_range_(const char* begin1, const char* end1)
234 {
235     size_t size1 = end1 - begin1;
236     reserve(size1 + 1);
237     std::copy(begin1, end1, end());
238     size_ += size1;
239     *end() = '\0';
240 }
```

4 src/string_view1.hxx

```
#pragma once

#include "Owned_string.hxx"
#include <iostream>

// Structure 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).
struct 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 the contents of a `String`.
    string_view(const Owned_string*);

    // Constructs a string_view from the contents of a `String`.
    string_view(Owned_string const&);

    // Constructs a string_view from a '\0'-terminated C-style string.
    explicit string_view(const char*);

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

    size_t size() const;

    /*
     * Member variables
     */

    const char* begin;
    const char* end;
};

// Overloads == for `string_view`s.
bool operator==(string_view, string_view);
```

```
43
44 // Overloads != for `string_view`s.
45 bool operator!=(string_view, string_view);
46
47 // Overloads stream insertion (printing)
48 std::ostream& operator<<(std::ostream&, string_view);
49
50 // Templates need to be defined in the .h file, not the .cpp file.
51 // (We subtract 1 from N because N will include the string literal's
52 // '\0' terminator.)
53 template <size_t N>
54 string_view::string_view(const char (&s) [N])
55         : string_view(s, N - 1)
56 { }
```

5 src/string_view1.cxx

```
1  #include "string_view1.hxx"
2  #include <algorithm>
3  #include <cstring>
4  #include <stdexcept>
5
6  string_view::string_view(const char* begin, const char* end)
7      : begin(begin)
8      , end(std::max(begin, end))
9  { }
10
11 string_view::string_view(const char* s, size_t size)
12     : string_view(s, s + size)
13 { }
14
15 string_view::string_view(const Owned_string* s)
16     : string_view(s->begin(), s->end())
17 { }
18
19 string_view::string_view(Owned_string const& s)
20     : string_view(s.begin(), s.end())
21 { }
22
23 string_view::string_view(const char* s)
24     : string_view(s, std::strlen(s))
25 { }
26
```

```
27 size_t string_view::size() const
28 {
29     return end - begin;
30 }
31
32 bool operator==(string_view sv1, string_view sv2)
33 {
34     return sv1.size() == sv2.size() &&
35         std::equal(sv1.begin, sv1.end, sv2.begin);
36 }
37
38 bool operator!=(string_view sv1, string_view sv2)
39 {
40     return !(sv1 == sv2);
41 }
42
43 std::ostream& operator<<(std::ostream& os, string_view sv)
44 {
45     return os.write(sv.begin, sv.size());
46 }
```

6 src/string_view2.hxx

```
1  #pragma once
2
3  #include "Owned_string.hxx"
4  #include <iostream>
5
6  // Class for representing ranges of characters, usually for the
7  // purpose of comparing them. The idea is that we have constructors for
8  // a bunch of different ways that strings might be specified, and each
9  // gets converted to a string_view [begin, end).
10 //
11 // Enforces invariant that begin <= end; throws otherwise.
12 class string_view
13 {
14 public:
15     // Constructs a string_view from `begin` and `end` directly.
16     string_view(const char* begin, const char* end);
17
18     // Constructs a string_view from the start and the size.
19     string_view(const char*, size_t);
20
```

```
21      // Constructs a string_view from the contents of a `String`.
22      string_view(Owned_string const&);
23
24      // Constructs a string_view from a '\0'-terminated C-style string.
25      explicit string_view(const char*);
26
27      // Constructs a string_view from a string literal using its static size.
28      template <size_t N>
29      string_view(const char (&s) [N]);
30
31      /*
32       * Getters
33       */
34
35      const char* begin() const;
36      const char* end() const;
37
38      size_t size() const;
39
40      /*
41       * Member variables
42       */
43
44  private:
45      const char* begin_;
46      const char* end_;
47      // INVARIANT: begin_ <= end_
48  };
49
50  // Overloads == for `string_view`s.
51  bool operator==(string_view, string_view);
52
53  // Overloads != for `string_view`s.
54  bool operator!=(string_view, string_view);
55
56  // Templates need to be defined in the .h file, not the .cpp file.
57  // (We subtract 1 from N because N will include the string literal's
58  // '\0' terminator.)
59  template <size_t N>
60  string_view::string_view(const char (&s) [N])
61          : string_view(s, N - 1)
62  { }
63
64  // Overloads stream insertion (printing)
65  std::ostream& operator<<(std::ostream&, string_view);
```

7 src/string_view2.cxx

```
#include "string_view2.hxx"
#include <algorithm>
#include <cstring>

string_view::string_view(const char* begin, const char* end)
    : begin_(begin), end_(end)
{
    if (begin > end)
        throw std::invalid_argument("string_view: begin > end");
}

string_view::string_view(const char* s, size_t size)
    : string_view(s, s + size)
{ }

string_view::string_view(Owned_string const& s)
    : string_view(s.begin(), s.end())
{ }

string_view::string_view(const char* s)
    : string_view(s, std::strlen(s))
{ }

size_t string_view::size() const
{
    return end_ - begin_;
}

const char* string_view::begin() const
{
    return begin_;
}

const char* string_view::end() const
{
    return end_;
}

bool operator==(string_view sv1, string_view sv2)
{
    return sv1.size() == sv2.size() &&
        std::equal(sv1.begin(), sv1.end(), sv2.begin());
```

```
}

bool operator!=(string_view sv1, string_view sv2)
{
    return !(sv1 == sv2);
}

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

8 test/test_Owned_string.cxx

```
#include "Owned_string.hxx"
#include "string_view1.hxx"

#include <catch.hxx>

#include <algorithm>
#include <cstring>

// Overloads == for `Owned_string`s.
static bool operator==(Owned_string const& s1, Owned_string const& s2)
{
    return string_view(s1) == string_view(s2);
}

// Tests that default construction works as expected.
TEST_CASE("default construction")
{
    Owned_string s;
    CHECK(s.empty());
    CHECK(s.size() == 0);
    CHECK(s == "");
}

// Tests the c_str constructor on the given C string `cs`.
static void c_str_constructor_case(const char* cs)
{
    Owned_string s(cs);
    CHECK(s.empty() == (cs[0] == '\0'));
    CHECK(s.size() == std::strlen(cs));
```

```
30      CHECK(s == cs);
31  }
32
33  // Test cases for `Owned_string_construct_c_str`.
34  TEST_CASE("c string constructor")
35  {
36      c_str_constructor_case("");
37      c_str_constructor_case("hello world");
38      c_str_constructor_case("hello\0world");
39  }
40
41  // Tests the range constructor on the given range [cp, cp + len).
42  static void range_constructor_case(string_view r)
43  {
44      Owned_string s(r.begin, r.end);
45      CHECK(s.empty() == (r.size() == 0));
46      CHECK(s.size() == r.size());
47      CHECK(s == r);
48  }
49
50  // Test cases for `Owned_string_construct_range`.
51  TEST_CASE("range constructor")
52  {
53      range_constructor_case("");
54      range_constructor_case("hello world");
55      range_constructor_case("hello\0world");
56      range_constructor_case("hello\0world");
57      range_constructor_case("hello\0world");
58  }
59
60  // Tests the copy constructor by copying a string that is first
61  // constructed from the range [cp, cp + len).
62  static void copy_constructor_case(string_view r)
63  {
64      Owned_string s1(r.begin, r.end);
65      Owned_string s2(s1);
66
67      CHECK(s1 == r);
68      CHECK(s2 == r);
69      CHECK(s1 == s2);
70
71      if (r.size() > 0) CHECK(s1.c_str() != s2.c_str());
72  }
73
74  // Tests the copy constructor.
```

```
75  TEST_CASE("copy constructor")
76  {
77      copy_constructor_case("");
78      copy_constructor_case("hello");
79      copy_constructor_case("hello\0world");
80  }
81
82  // Tests the move constructor by moving a string that is first
83  // constructed from the range [cp, cp + len).
84  static void move_constructor_case(string_view r)
85  {
86      Owned_string s1(r.begin, r.end);
87      CHECK(s1 == r);
88
89      Owned_string s2(std::move(s1));
90      CHECK(s1 == "");
91      CHECK(s2 == r);
92  }
93
94  // Tests the move constructor.
95  TEST_CASE("move constructor")
96  {
97      move_constructor_case("");
98      move_constructor_case("hello");
99      move_constructor_case("hello\0world");
100 }
101
102 // Tests the copy-assignment operator by constructing strings
103 // from the ranges [cs1, cs1 + len1) and [cs2, cs2 + len2), and
104 // then copy-assigning the latter to the former.
105 static void copy_assignment_case(string_view r1, string_view r2)
106 {
107     Owned_string s1(r1.begin, r1.end);
108     Owned_string s2(r2.begin, r2.end);
109     const char* old1 = s1.c_str();
110     s1 = s2;
111     CHECK(s1 == s2);
112     const char* new1 = s1.c_str();
113     CHECK((r1.size() >= r2.size()) == (new1 == old1));
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.begin[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.begin, r.end);
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(string_view(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(string_view(s) == "b\0rfoo");
246 }
```

9 test/test_string_view1.cxx

```
1 #include "string_view1.hxx"
2 #include <catch.hxx>
3 #include <cstring>
```

9 test/test_string_view1.cxx

```
4  #include <sstream>
5
6  static const char* const hello0world = "hello\0world";
7
8  TEST_CASE("range constructor")
9  {
10     string_view sv1 {hello0world, hello0world + std::strlen(hello0world)};
11     string_view sv2 {hello0world, hello0world + 11};
12     CHECK(sv1.size() == 5);
13     CHECK(sv2.size() == 11);
14 }
15
16 TEST_CASE("crossed range becomes empty")
17 {
18     string_view sv {hello0world + 3, hello0world};
19     CHECK( sv.size() == 0 );
20 }
21
22 TEST_CASE("start and size constructor")
23 {
24     string_view sv1 {hello0world, std::strlen(hello0world)};
25     string_view sv2 {hello0world, 11};
26     CHECK(sv1.size() == 5);
27     CHECK(sv2.size() == 11);
28 }
29
30 TEST_CASE("C style")
31 {
32     string_view sv {hello0world};
33     CHECK(sv.size() == 5);
34 }
35
36 TEST_CASE("stream insertion (printing)")
37 {
38     string_view sv1("hello");
39     string_view sv2(hello0world, 11);
40
41     std::ostringstream oss;
42     oss << sv1;
43     CHECK( oss.str() == "hello" );
44
45     oss.str("");
46     oss << sv2;
47     CHECK( oss.str() == std::string(hello0world, 11) );
48 }
```

10 test/test_string_view2.cxx

```
1  #include "string_view2.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, hello0world + 11);
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  }
```