```
#pragma once

#include "Owned_string.h"
#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);

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

// Overloads stream insertion (printing)
std::ostream& operator<<(std::ostream&, string_view);

// Templates need to be defined in the .h file, not the .cpp file.
// (We subtract 1 from N because N will include the string literal's
// '\0' terminator.)
template <size_t N>
string_view::string_view(const char (&s) [N])
        : string_view(s, N - 1)
{ }
```

```cpp
#include "string_view1.h"
#include <algorithm>
#include <cstring>
#include <stdexcept>

string_view::string_view(const char* begin, const char* end)
    : begin(begin)
    , end(std::max(begin, end))
{ }

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

string_view::string_view(const Owned_string* s)
    : string_view(s->begin(), s->end())
{ }

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;
}

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());
}
```

```cpp
#include "string_view1.h"
#include <catch.h>
#include <cstring>
#include <sstream>

static const char* const hello0world = "hello\0world";

TEST_CASE("range constructor")
{
    string_view sv1 {hello0world, hello0world + std::strlen(hello0world)};
    string_view sv2 {hello0world, hello0world + 11};
    CHECK(sv1.size() == 5);
    CHECK(sv2.size() == 11);
}

TEST_CASE("crossed range becomes empty")
{
    string_view sv {hello0world + 3, hello0world};
    CHECK( sv.size() == 0 );
}

TEST_CASE("start and size constructor")
{
    string_view sv1 {hello0world, std::strlen(hello0world)};
    string_view sv2 {hello0world, 11};
    CHECK(sv1.size() == 5);
    CHECK(sv2.size() == 11);
}

TEST_CASE("C style")
{
    string_view sv {hello0world};
    CHECK(sv.size() == 5);
}

TEST_CASE("stream insertion (printing)")
{
    string_view sv1("hello");
    string_view sv2(hello0world, 11);

    std::ostringstream oss;
    oss << sv1;
    CHECK( oss.str() == "hello" );

    oss.str("");
    oss << sv2;
    CHECK( oss.str() == std::string(hello0world, 11) );
}
```

```
#pragma once

#include "Owned_string.h"
#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 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(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]);

    /*
     * Getters
     */

    const char* begin() const;
    const char* end() const;

    size_t size() const;

    /*
     * Member variables
     */

private:
    const char* begin_;
    const char* end_;
    // INVARIANT: begin_ <= end_
};

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

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

// Templates need to be defined in the .h file, not the .cpp file.
// (We subtract 1 from N because N will include the string literal's
// '\0' terminator.)
template <size_t N>
string_view::string_view(const char (&s) [N])
        : string_view(s, N - 1)
{ }

// Overloads stream insertion (printing)
std::ostream& operator<<(std::ostream&, string_view);
```

```cpp
#include "string_view2.h"
#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());
}
```

```cpp
#include "string_view2.h"
#include "Owned_string.h"
#include <catch.h>
#include <sstream>

static const char hello[]       = "hello";
static const char hello0world[] = "hello\0world";

TEST_CASE("invariant")
{
    string_view(hello, hello + 3);
    CHECK_THROWS_AS(string_view(hello + 3, hello), std::invalid_argument);
}

TEST_CASE("stream insertion (printing)")
{
    string_view sv1("hello");
    string_view sv2(hello0world, 11);

    std::ostringstream oss;
    oss << sv1;
    CHECK( oss.str() == "hello" );

    oss.str("");
    oss << sv2;
    CHECK( oss.str() == std::string(hello0world, 11) );
}

TEST_CASE("construction")
{
    Owned_string s3(hello0world, hello0world + 11);

    // Constructing from [begin, end) range:

    string_view r11(hello, hello + 5);
    CHECK(r11.size() == 5);

    string_view r12(std::begin(hello0world), std::end(hello0world) - 1);
    CHECK(r12.size() == 11);

    string_view r13(s3);
    CHECK(r13.size() == 11);

    // Constructing from start, length:

    string_view r21(hello, 5);
    CHECK(r21.size() == 5);

    string_view r22(hello0world, sizeof hello0world - 1);
    CHECK(r22.size() == 11);

    string_view r23(s3.c_str(), s3.size());
    CHECK(r23.size() == 11);

    // Constructing from sized array or `String`:

    string_view r31 = hello;
    CHECK(r31.size() == 5);

    string_view r32 = hello0world;
    CHECK(r32.size() == 11);

    string_view r33 = s3;
    CHECK(r33.size() == 11);
}
```

```
#pragma once

#include <cstddef>

// 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 (â\200\234member variableâ\200\235Â is
// C++-speak for â\200\234fieldâ\200\235) 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.
struct Owned_string
{
    /*
     * 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.
     */

    // Initializes `this` to the empty string.
    Owned_string();

    // Initializes `this` to a copy of the given C ('\0'-terminated)
    // string.
    Owned_string(const char*);

    // Initializes `this` to a copy of the array starting at `begin` and
    // ending at (not including) `end`. This array may contain '\0's, and
    // they will be copied too.
    Owned_string(const char* begin, const char* end);

    // Copy constructor: initializes `this` to be a copy of the
    // argument.
    Owned_string(Owned_string const&);

    // Move constructor: initializes `this` by stealing the argument's
    // memory, leaving it valid but empty.
    Owned_string(Owned_string&&) noexcept;

    /*
     * Destructor.
     */
```

```
    // C++ calls this automatically whenever a `Owned_string` object needs to
    // be destroyed (such as when it goes out of scope).
    ~Owned_string();

    /*
     * Assignment â\200\234operatorsâ\200\235.
     */

    // Assigns the contents of the argument to `this`. This may reuse
    // `this`'s memory or reallocate.
    Owned_string& operator=(Owned_string const&);

    // Steals the argument's memory, assigning it to `this` and leaving
    // the argument object valid but empty.
    Owned_string& operator=(Owned_string&&) noexcept;

    /*
     * Non-lifecycle `Owned_string` operations.
     */

    // Returns whether `this` is the empty string.
    bool empty() const;

    // Returns the number of characters in `this`. This does not
    // depend on '\0' termination, and internal '\0's are allowed.
    size_t size() const;

    // Returns the character of `this` at the given index.
    //
    // ERROR: If `index >= this->size()` then the behavior is
    // undefined.
    char operator[](size_t index) const;

    // Returns a reference to the character of `this` at the given index.
    //
    // ERROR: If `index >= this->size()` then the behavior is
    // undefined.
    char& operator[](size_t index);

    // Adds character `c` to the end of `this`. This may cause pointers
    // returned by previous calls to `this->operator[]` or `this->c_str`
    // to become invalidated.
    void push_back(char c);

    // Removes the last character of the string.
    //
    // ERROR: UB if `Owned_string(this)`.
    void pop_back();

    // Expands the capacity, if necessary, to hold `additional_cap` more
    // characters.
    void reserve(size_t additional_cap);

    // Swaps the contents of two strings without copying the actual
    // characters.
    void swap(Owned_string&);

    // Appends another string onto this string.
    Owned_string& operator+=(Owned_string const& that);

    // Appends a C string onto this string.
    Owned_string& operator+=(const char* that);

    // Returns a pointer to the content of this string as a C-style string.
```

```
    // Note that internal '\0's will make `strlen(String_c_str(s))` less than
    // `String_size(s)`, which doesn't depend on the content of the string.
    const char* c_str() const;

    /*
     * Some operations that were hard to define in lec09-String.h-style.
     */

    // Returns a pointer to the first character of the string.
    const char* begin() const;
    char* begin();

    // Returns a pointer to the first character past the end of the
    // string.
    const char* end() const;
    char* end();

    /*
     * PRIVATE HELPER FUNCTIONS.
     *
     * These are functions that are useful for implementing the
     * functions above. They should not be called by clients.
     * Thus, they are documented in the implementation file, not
     * in this header.
     */

private:
    void set_empty_();
    void ensure_capacity_(size_t min_cap);
    void append_range_(const char* begin, const char* end);

    /*
     * Data members -- clients can't touch these.
     *
     * These are where the data is actually stored---everything above
     * here declares operations (member functions), and below is data.
     */

    size_t size_;
    size_t capacity_;
    char*  data_;
};

/*
 * These are free functions. Some of them *could* be defined as members,
 * but it improves encapsulation to minimize the number of members of
 * possible.
 */

// Appends two strings, returning a new string.
Owned_string operator+(Owned_string const&, Owned_string const&);

// Appends two strings, returning a new string.
Owned_string operator+(Owned_string const&, const char*);

// Appends two strings, returning a new string.
Owned_string operator+(const char*, Owned_string const&);

// Steals the left string's memory to append the right and return the
// result.
Owned_string operator+(Owned_string&&, Owned_string const&);

// Steals the left string's memory to append the right and return the
// result.
Owned_string operator+(Owned_string&&, const char*);
```