Access Control

CS 211

Winter 2020

Road map

- A borrowed string view type
- Adding access control

A borrowed string view type

A borrowed string view type

We can use pointer ranges to represent borrowed strings:

```
struct string_view
{
    const char* begin;
    const char* end;
};
```

A borrowed string view type

We can use pointer ranges to represent borrowed strings:

```
struct string_view
{
    const char* begin;
    const char* end;
};

TEST_CASE("constructing␣a␣string_view")
{
    const char s[] = "hello\0world";
    string_view sv1 {s, s + std::strlen(s)};
    CHECK( sv1.end - sv1.begin == 5 );



}
```

A borrowed string view type

We can use pointer ranges to represent borrowed strings:

```
struct string_view
{
    const char* begin;
    const char* end;
};

TEST_CASE("constructing␣a␣string_view")
{
    const char s[] = "hello\0world";
    string_view sv1 {s, s + std::strlen(s)};
    CHECK( sv1.end - sv1.begin == 5 );

    string_view sv2 {s, s + sizeof s - 1};
    CHECK( sv2.end - sv2.begin == 11 );
}
```

Adding a member function

In C++, `struct` members are not only variables. Here we add a member function:

```
struct string_view
{
    size_t size() const;    // function member

    const char* begin;      // data member
    const char* end;        // data member
};
```

Adding a member function

In C++, `struct` members are not only variables. Here we add a member function:

```
struct string_view
{
    size_t size() const;    // function member

    const char* begin;      // data member
    const char* end;        // data member
};

TEST_CASE("string_view::size()␣const")
{
    const char* s = "hello\0world";
    string_view sv {s, s + 11};
    CHECK( sv.size() == 11 );
}
```

Why a member function?

Why not this?:

```
size_t size(string_view sv)
{
    return sv.end - sv.begin;
}
```

Why a member function?

Why not this?:

```
size_t size(string_view sv)
{
    return sv.end - sv.begin;
}
```

Special things members can do:

- access other, private members (we'll see this soon)
- override lifecycle operations (we'll see this soon)
- not really nice having global function named `size`

How do we define a member function?

Member function definitions:

- Have their names prefixed by *Type*`::`
- Take an implicit parameter *Type*`* this` or `const` *Type*`* this`

```
size_t string_view::size() const
{
    // `this` has type `const string_view*`
    return this->end - this->begin;
}
```

How do we define a member function?

Member function definitions:

- Have their names prefixed by *Type*`::`
- Take an implicit parameter *Type*`* this` or `const` *Type*`* this`

```
size_t string_view::size() const
{
    // `this` has type `const string_view*`
    return end - begin;
}
```

Also, `this->` is implicit on member names!

Aside: Member access syntax

What is the difference between `thing.member` and `thing::member`?

Aside: Member access syntax

What is the difference between `thing.member` and `thing::member`?

- `Type::member` access a member of a type (struct or class)
- `instance.member` access a member of a value

Aside: Member access syntax

What is the difference between `thing.member` and `thing::member`?

- `Type::member` access a member of a type (struct or class)
- `instance.member` access a member of a value

Examples:

- `string_view::size` names the `size` member function of the `string_view` type in general
- `an_sv.size` means the `size` member function on a particular instance of `string_view` (`an_sv`)
- `an_sv.begin` means the `begin` member variable of a particular instance of `string_view` (`an_sv`)
- `string_view::begin` (usually) doesn't mean anything

Operator overloading

We can tell C++ the meaning of operators (like == and +) for our types.

Declaration (goes in .hxx):

```
bool operator==(string_view, string_view);
```

Definition (goes in .cxx):

```
#include <algorithm>

bool operator==(string_view a, string_view b)
{
    return a.size() == b.size() &&
        std::equal(a.begin, a.end, b.begin);
}
```

More operator overloading

We can also make our new type printable.

Declaration (goes in .hxx):

```
std::ostream& operator<<(std::ostream&, string_view);
```

Definition (goes in .cxx):

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

Making construction more convenient

A constructor is:

- a member function
- with no result type
- whose name is the same as the name of the struct.

If you declare constructors then all object creation goes via the constructor. For example:

Making construction more convenient

A constructor is:

- a member function
- with no result type
- whose name is the same as the name of the struct.

If you declare constructors then all object creation goes via the constructor. For example:

```
struct string_view
{
    string_view(const char* start, size_t size);
    const char *begin, *end;
};
```

Making construction more convenient

A constructor is:

- a member function
- with no result type
- whose name is the same as the name of the struct.

If you declare constructors then all object creation goes via the constructor. For example:

```
struct string_view
{
    string_view(const char* start, size_t size);
    const char *begin, *end;
};

const char* s = "hello";
string_view sv {s, s + 5}; // error: no match
string_view sv {s, 5};     // all good
```

How does that make it more convenient though?

Multiple constructors, chosen by argument type:

```
struct string_view
{
    string_view(const char* begin, const char* end);
    string_view(const char* start, size_t size);
    string_view(const char* c_str);
    string_view(String const& s);
    ...
};
```

How does that make it more convenient though?

Multiple constructors, chosen by argument type:

```
struct string_view
{
    string_view(const char* begin, const char* end);
    string_view(const char* start, size_t size);
    string_view(const char* c_str);
    string_view(String const& s);
    ...
};

const char* s1 = "hello\0world";
String s2(s1, s1 + 11);      // future constructor
```

How does that make it more convenient though?

Multiple constructors, chosen by argument type:

```
struct string_view
{
    string_view(const char* begin, const char* end);
    string_view(const char* start, size_t size);
    string_view(const char* c_str);
    string_view(String const& s);
    ...
};

const char* s1 = "hello\0world";
String s2(s1, s1 + 11);        // future constructor
string_view sv1(s1, s1 + 11); // 1st constructor
string_view sv2(s1, 11);      // 2nd constructor
string_view sv3(s1);          // 3rd constructor
string_view sv4(s2);          // 4th constructor
```

Defining constructors

Constructors have a special syntax for initializing member variables:

```
string_view::string_view(const char* begin0,
                         const char* end0)
        : begin(begin0)
        , end(end0)
{ } // <= regular function body, often left empty
```

Defining constructors

Constructors have a special syntax for initializing member variables:

```
string_view::string_view(const char* begin0,
                         const char* end0)
        : begin(begin0)
        , end(end0)
{ } // <= regular function body, often left empty
```

Constructors can also delegate to other constructors:

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

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

Constructors can enforce invariants

Suppose we decide that a valid `string_view` should never has a negative size.

C++ can help us *guarantee* this for all `string_views`.

Constructors can enforce invariants

Suppose we decide that a valid `string_view` should never has a negative size.

C++ can help us *guarantee* this for all `string_view`s.

The first step is to avoid constructing invalid `string_view`s.

Constructors can enforce invariants

Suppose we decide that a valid `string_view` should never has a negative size.

C++ can help us *guarantee* this for all `string_views`.

The first step is to avoid constructing invalid `string_views`. We could fix improper ranges:

```
string_view::string_view(const char* begin0,
                         const char* end0)
        : begin(begin0)
        , end(std::max(begin0, end0))
{ }
```

Constructors can enforce invariants

Suppose we decide that a valid `string_view` should never has a negative size.

C++ can help us *guarantee* this for all `string_view`s.

The first step is to avoid constructing invalid `string_view`s. Or we could reject improper ranges:

```
string_view::string_view(const char* begin0,
                         const char* end0)
        : begin(begin0)
        , end(end0)
{
    if (end0 < begin0)
        throw std::invalid_argument(BAD_RANGE);
}
```

This ensures we never construct an invalid `string_view`.

Okay, but what if I…?

Okay, but what if I…?

```
const char* s = "hello";
string_view sv(s);
```

Okay, but what if I…?

```
const char* s = "hello";
string_view sv(s);
sv.end = sv.begin - 3;
```

Okay, but what if I…?

```
const char* s = "hello";
string_view sv(s);
sv.end = sv.begin - 3;
```

Oh no!

Member access control

New idea: Access modifiers

With access modifiers, we can control exactly what client code is allowed to do with our struct:

```
struct Name
{
    // visible to all

private:
    // visible only to other members

public:
    // visible to all
}
```

Introducing `classes`

Technically, `classes` and `structs` differ only in their default access modifier:

- `class T { ... };` $\equiv$ `struct T { private: ... };`
- `struct T { ... };` $\equiv$ `class T { public: ... };`

Introducing `classes`

Technically, `classes` and `structs` differ only in their default access modifier:

- `class T { ... };` $\equiv$ `struct T { private: ... };`
- `struct T { ... };` $\equiv$ `class T { public: ... };`

But in connotation, we will use `class` for “smart data” and `struct` for “plain old data.”

Plan for encapsulation

1. Make member variables `private`
2. Add public member functions to let clients access what we want them to access
3. Don't add public member functions that let clients do bad things

A `string_view` class

```
class string_view
{
public:
    // Constructors:
    string_view(const char*, const char*);
    string_view(String const&);
    ...

    // Accessors:
    const char* begin() const;
    const char* end() const;

private:
    const char *begin_, *end_;
};
```

Implementing the accessors

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

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

Non-member functions must use accessors

Doesn't work because `string_view::begin_` and `string_view::end_` are `private`:

```
bool operator==(string_view a, string_view b)
{
    return a.size() == b.size() &&
        std::equal(a.begin_, a.end_, b.begin_);
}
```

Non-member functions must use accessors

Works because `string_view::begin()` and `string_view::end()` are `public`:

```
bool operator==(string_view a, string_view b)
{
    return a.size() == b.size() &&
        std::equal(a.begin(), a.end(), b.begin());
}
```

Non-member functions must use accessors

Works because `string_view::begin()` and `string_view::end()` are `public`:

```
bool operator==(string_view a, string_view b)
{
    return a.size() == b.size() &&
        std::equal(a.begin(), a.end(), b.begin());
}
```

This is a *good thing*, because it means that non-members can't break our carefully preserved invariants

Welcome to encapsulation!

Encapsulation is a software engineering principle that says:

1. Bundle your data and your operations together
2. Don’t let non-bundled operations mess with your bundled data

Welcome to encapsulation!

Encapsulation is a software engineering principle that says:

1. Bundle your data and your operations together
2. Don't let non-bundled operations mess with your bundled data

Benefits:

- Correctness: only your operations are responsible for preserving invariants, because clients cannot mess them up
- Flexibility: you can change details of the implementation without changing clients, provided the API remains the same

Example of flexibility

Client code can't distinguish this from the previous version:

```
class string_view
{
public:
    string_view(const char*, const char*);
    string_view(const char*, size_t);
    ...

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

private:
    const char* start_;
    size_t size_;
};
```

– Next: RAII! —

– Next: RAIII!!I! —