Access Control for Encapsulation

CS 211/Spring 2020
Road map

A borrowed string view struct
  Operator overloading
  Constructors for safety & convenience
  The problem

Goal: Protecting invariants
  Mechanism: Member access control
  Technique: Encapsulation
A borrowed string view struct
A nicer borrowed string representation

Instead of relying on a terminating 0 character, we can represent a borrowed string using a pointer range:

```c
struct string_view
{
    char const* begin;
    char const* end;
};
```
A nicer borrowed string representation

Instead of relying on a terminating 0 character, we can represent a borrowed string using a pointer range:

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

```cpp
test_case("constructing a string_view using strlen")
{
    char const s[] = "hello\0world";
    string_view sv {s, s + std::strlen(s)};
    CHECK( sv.end - sv.begin == 5 );
}
```
A nicer borrowed string representation

Instead of relying on a terminating 0 character, we can represent a borrowed string using a pointer range:

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

```cpp
test_case("constructing a string_view using sizeof")
{
    char const s[] = "hello\0world";
    string_view sv {s, s + sizeof s - 1};
    CHECK( sv.end - sv.begin == 11 );
}
```
A nicer borrowed string representation

Instead of relying on a terminating 0 character, we can represent a borrowed string using a pointer range:

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

```c
TEST_CASE("careful...")
{
    char const s[] = "hello\0world";
    string_view sv {s, s + sizeof(s - 1)};
    CHECK( sv.end - sv.begin == 8 );
}
```
A nicer borrowed string representation

Instead of relying on a terminating 0 character, we can represent a borrowed string using a pointer range:

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

```c
TEST_CASE("slicing")
{
    char const s[] = "hello\0world";
    string_view sv1 {s, s + sizeof(s - 1)};
    string_view sv2 {sv1.begin + 6, sv1.end};
    CHECK( sv2.size() == 5 );
}
```
Adding a member function

In C++, \texttt{struct} members are not only variables. Here we add a member function:

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

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

```cpp
TEST_CASE("string_view::size() const")
{
    char const* s = "hello world";
    string_view sv {s, s + 11};
    CHECK(sv.size() == 11);
}
```
Adding a member function

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

```cpp
struct string_view
{
    size_t size() const;  // function member
    char const* begin;   // data member
    char const* end;     // data member
};
```

```cpp
TEST_CASE("string_view::size() const")
{
    char const* s = "hello\0world";
    string_view sv {s, s + 11};
    CHECK( sv.size() == 11 );
}
```
Why a member function?

Why not this?:

```cpp
size_t size(string_view sv)
{
    return sv.end - sv.begin;
}
```
Why a member function?

Why not this?:

```cpp
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(s) named `size`
Operator overloading
Defining our own equality

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

// Equality for `string_view`'s.
bool operator==(string_view, string_view);
Defining our own equality

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

```cpp
// Equality for 'string_view's.
bool operator==(string_view, string_view);

#include "string_view.hxx"
#include <algorithm>

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

Defining our own formatting function

We can also make our new type printable.

Declaration (goes in .hxx):

```cpp
#include <iostream>

// Stream insertion for 'string_view'.
std::ostream& operator<<(std::ostream&, string_view);
```
Defining our own formatting function

We can also make our new type printable.

Declaration (goes in .hxx):

```cpp
#include <iostream>
#include "string_view.hxx"

// Stream insertion for `string_view`.
std::ostream& operator<<(std::ostream&, string_view);

#include "string_view.cxx"

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

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:

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

char const* s = "hello";
string_view sv {s, s + 5}; // error: doesn’t match constructor
string_view sv {s, 5}; // all good: uses constructor
```
Review: Constructors

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:

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

char const* s = "hello";
string_view sv {s, s + 5};
```
Review: Constructors

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:

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

char const* s = "hello";
string_view sv {s, s + 5}; // error: doesn't match constructor
```
Review: Constructors

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:

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

char const* s = "hello";
string_view sv {s, s + 5}; // error: doesn't match constructor
string_view sv {s, 5};  // all good: uses constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);

    ...  

};
```

```cpp
std::string hi("
  hello"
);

string_view sv1(hi);
// 1st constructor

string_view sv2(sv1.begin, sv1.end);
// 2nd constructor

string_view sv3(hi.data(), hi.size());
// 3rd constructor

string_view sv4;
// 4th (default) constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);

    ...
};
```

```cpp
std::string hi("
hello"
);
string_view sv1(hi);
// 1st constructor
string_view sv2(sv1.begin, sv1.end);
// 2nd constructor
string_view sv3(hi.data(), hi.size);
// 3rd constructor
string_view sv4;
// 4th (default) constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    ...
};
```

```cpp
std::string hi("
hello"
); string_view sv1(hi); // 1st constructor string_view sv2(sv1.begin, sv1.end); // 2nd constructor string_view sv3(hi.data(), hi.size()); // 3rd constructor string_view sv4; // 4th (default) constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    string_view();
    ...
};
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    string_view();
    ...
};
```

```cpp
std::string hi("hello");
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    string_view();
    ...
};
```

```cpp
std::string hi("hello");
string_view sv1(hi); // 1st constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view {
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    string_view();
    ...
};
```

```cpp
std::string hi("hello");
string_view sv1(hi); // 1st constructor
string_view sv2(sv1.begin, sv1.end); // 2nd constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view
{
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    string_view();
    ...
};
```

```cpp
std::string hi("hello");
string_view sv1(hi);          // 1st constructor
string_view sv2(sv1.begin, sv1.end);  // 2nd constructor
string_view sv3(hi.data(), hi.size()); // 3rd constructor
```
How does that make it more convenient though?

Multiple constructors (overloaded by argument type):

```cpp
struct string_view {
    string_view(std::string const& s);
    string_view(char const* begin, char const* end);
    string_view(char const* start, size_t size);
    string_view();
    ...
};
```

```cpp
std::string hi("hello");
string_view sv1(hi); // 1st constructor
string_view sv2(sv1.begin, sv1.end); // 2nd constructor
string_view sv3(hi.data(), hi.size()); // 3rd constructor
string_view sv4; // 4th (default) constructor
```
Defining constructors

Constructors have a special syntax for initializing member variables:

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

Constructors have a special syntax for initializing member variables:

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

Constructors can also delegate to other overloads:

```cpp
string_view::string_view(char const* start, size_t size)
    : string_view(start, start + size)    // <= delegation
{ }
```

```cpp
string_view::string_view(char const* c_str)
    : string_view(c_str, std::strlen(c_str)) // <= delegation
{ }
```
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 have a negative size.

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

The first step is to avoid constructing invalid `string_views`. How?
Constructors can enforce invariants

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

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

The first step is to avoid constructing invalid `string_views`. How?

We could fix improper ranges:

```cpp
string_view::string_view(char const* begin0,
                         char const* end0)
  : begin(begin0),
    end(std::max(begin0, end0))
{}  
```
Constructors can enforce invariants

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

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

The first step is to avoid constructing invalid string_views. How?

Or we could reject improper ranges:

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

This ensures we never construct an invalid string_view.
The problem
Okay, but what if I...?

```cpp
std::string s("hello");
string_view sv(s);
sv.end = sv.begin - 3;
```
Okay, but what if I...?

```cpp
std::string s("hello");
string_view sv(s);
sv.end = sv.begin - 3;
```

Oh no! We need a way to control access.
Goal: Protecting invariants
Mechanism: Member access control
New idea: Access modifiers

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

```cpp
struct My_struct
{
    // visible to all functions

    // visible to all functions

    // visible to all functions

    // visible to all functions
};
```
New idea: Access modifiers

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

```cpp
struct My_struct
{
    // visible to all functions

private:
    // visible only to member functions of `My_struct`

    // visible only to member functions of `My_struct`

    // visible only to member functions of `My_struct`
};
```
New idea: Access modifiers

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

```cpp
struct My_struct {
    // visible to all functions

    private:
        // visible only to member functions of 'My_struct'

    public:
        // visible to all functions

    // visible to all functions
};
```
New idea: Access modifiers

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

```cpp
struct My_struct
{
    // visible to all functions

private:
    // visible only to member functions of 'My_struct'

public:
    // visible to all functions

private:
    // visible only to member functions of 'My_struct'
};
```
Introducing classes

Technically, classes and structs differ only in whether their members default to public or private:

class My_class
{
    // visible only to member functions of 'My_class'

public:
    // visible to all functions

private:
    // visible only to member functions of 'My_class'

public:
    // visible to all functions
};
Introducing classes

Technically, **classes** and **structs** differ only in whether their members default to **public** or **private**:

class My_class
{
    // visible only to member functions of 'My_class'

    public:
        // visible to all functions

    private:
        // visible only to member functions of 'My_class'

        // visible only to member functions of 'My_class'

};
Introducing classes

Technically, classes and structs differ only in whether their members default to public or private:

```cpp
class My_class
{
    // visible only to member functions of 'My_class'

public:
    // visible to all functions

    // visible to all functions

    // visible to all functions

};
```
Introducing classes

Technically, classes and structs differ only in whether their members default to public or private:

class My_class
{
    // visible only to member functions of 'My_class'

    // visible only to member functions of 'My_class'

    // visible only to member functions of 'My_class'

    // visible only to member functions of 'My_class'
};
Technically, they’re nearly interchangeable

\[
\text{struct } \{ \ldots \} ; \equiv \text{class } \{ \text{public: } \ldots \} ;
\]
Technically, they’re nearly interchangeable

\[
\text{struct \{ ... \}; } \equiv \text{class \{ public: ... \};}
\]

\[
\text{class \{ ... \}; } \equiv \text{struct \{ private: ... \};}
\]
Technically, they’re nearly interchangeable

```
struct { ... };
≡
class { public: ... };
```

```
class { ... };
≡
struct { private: ... };
```

But in connotation, we will use class for “smart data” and struct for “plain old data.”
Technique: Encapsulation
Introducing encapsulation

Encapsulation means making data private to its operations.

Here’s the plan
Introducing encapsulation

Encapsulation means making data private to its operations.

Here’s the plan:

1. Make member variables private
Introducing encapsulation

Encapsulation means making data private to its operations.

Here’s the plan:

1. Make member variables **private**
2. Add public member functions to let clients do useful things
Introducing encapsulation

Encapsulation means making data private to its operations.
Here’s the plan:

1. Make member variables \textit{private}
2. Add public member functions to let clients do useful things
3. Don’t add public member functions that let clients do bad things
A string_view class

class string_view
{
    char const *begin_, *end_;

public:
    string_view();
    string_view(std::string const &);
    string_view(char const*, char const*);
    :

    char const* begin() const;
    char const* end() const;
    :
};
A string_view class

class string_view
{
    char const *begin_, *end_;

public:
    string_view();
    string_view(std::string const&);
    string_view(char const*, char const*);
    ...

    char const* begin() const;
    char const* end() const;
    ...
};
A string_view class

class string_view
{
    char const *begin_, *end_;  

public:
    string_view();
    string_view(std::string const&);
    string_view(char const*, char const*);

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

};
A string_view class

class string_view
{
    char const *begin_, *end_;  // Private member variables

public:
    string_view();  // Public constructors
    string_view(std::string const&);
    string_view(char const*, char const*);
    ...;

    char const* begin() const;
    char const* end() const;
    ...;
};
A `string_view` class

class string_view
{
    char const *begin_, *end_;  

public:
    string_view();
    string_view(std::string const&);
    string_view(char const*, char const*);
    ...

    char const* begin() const;
    char const* end() const;
    ...
};
Implementing the accessors

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

cchar const* string_view::end() const
{
    return end_;  
}
Non-member functions must use accessors now

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

This doesn't work because `string_view::begin_` and `string_view::end_` are private and `operator==(string_view, string_view)` isn't a member.
Non-member functions must use accessors now

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

Doesn’t work because `string_view::begin_` and `string_view::end_` are private and `operator==(string_view, string_view)` isn’t a member

This is a good thing, because it means that non-members can’t break our carefully preserved invariants.
Non-member functions must use accessors now

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

Works because string_view::begin() and string_view::end() are public
```
Non-member functions must use accessors now

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

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

Now client (non-member) functions can see the values, but cannot freely change them.
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
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
{
    char const* start_;  
    size_t     size_;  

public:
    string_view();
    string_view(std::string const&);
    string_view(char const*, char const*);
    char const* begin() const;
    char const* end() const;
    ;
};
– Next: Game Physics –
Member functions
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`

```cpp
size_t string_view::size() const
{
    // 'this' has type 'string_view const*'
    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`

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

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

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