The C++ Object Lifecycle

EECS 211
Winter 2019
Initial code setup

$ cd eecs211
$ curl $URL211/lec/09lifecycle.tgz | tar zx
...
$ cd 09lifecycle
Road map

- An owned string type
- A borrowed string type
An owned string type
Our own String type

This is C++:

```cpp
struct String {
    char* data_;  
    size_t size_, capacity_; 
};
```
Our own String type

This is C++:

```cpp
struct String
{
    char* data_;  // points to the string data (array of characters)
    size_t size_, capacity_;
};
```

The idea:

- `data_` points to the string data (array of characters)
Our own String type

This is C++:

```cpp
struct String {
    char* data_;  // points to the string data (array of characters)
    size_t size_, capacity_;  // actual number of characters in our string (we don't rely on '\0' termination anymore)
};
```

The idea:

- `data_` points to the string data (array of characters)
- `size_` is the actual number of characters in our string (we don’t rely on '\0' termination anymore)
Our own String type

This is C++:

```cpp
struct String
{
    char* data_;  
    size_t size_, capacity_;  
};
```

The idea:

- `data_` points to the string data (array of characters)
- `size_` is the actual number of characters in our string (we don’t rely on '\0' termination anymore)
- `capacity_` is the allocated size of `data_`, which might exceed `size_`, giving us space to grow
Our own String type

This is C++:

```cpp
struct String
{
    char* data_;  
    size_t size_, capacity_;  
};
```

The idea:

- `data_` points to the string data (array of characters)
- `size_` is the actual number of characters in our string (we don’t rely on ‘\0’ termination anymore)
- `capacity_` is the allocated size of `data_`, which might exceed `size_`, giving us space to grow
- (We ‘\0’-terminate anyway to facilitate interaction with C—but note also that internal ‘\0’s will make C not see the whole string)
Our own String type

This is C++:

```cpp
struct String
{
  char* data_;  
  size_t size_, capacity_;  
};
```

Invariants (must always be true for String s to be valid):
Our own String type

This is C++:

```cpp
struct String {
    char* data_;  
    size_t size_, capacity_;  
};
```

Invariants (must always be true for String s to be valid):

1. `s.capacity_ == 0` if and only if `s.data_ == nullptr`
Our own String type

This is C++:

```cpp
struct String
{
    char* data_;  // Should be unique and free store
    size_t size_, capacity_;  // Must be initialized

};
```

Invariants (must always be true for String s to be valid):

1. `s.capacity_ == 0` if and only if `s.data_ == nullptr`
2. If `s.capacity_ > 0` then:
   1. `s.data_[0]`, …, `s.data_[s.size_ - 1]` are initialized
   2. `s.data_[s.size_] == \0`

5
Our own String type

This is C++:

```cpp
struct String {
    char* data_;  
    size_t size_, capacity_; 
};
```

Invariants (must always be true for String `s` to be valid):

1. `s.capacity_ == 0` if and only if `s.data_ == nullptr`
2. If `s.capacity_ > 0` then:
   2.1 `s.data_` points to a unique, free store–allocated array of `s.capacity_` chars
Our own String type

This is C++:

```cpp
struct String {
    char* data_;  
    size_t size_, capacity_; 
};
```

Invariants (must always be true for String s to be valid):

1. `s.capacity_ == 0` if and only if `s.data_ == nullptr`
2. If `s.capacity_ > 0` then:
   2.1 `s.data_` points to a unique, free store-allocated array of `s.capacity_` chars
   2.2 `s.size_ + 1 <= s.capacity_`
Our own String type

This is C++:

```cpp
struct String
{
    char*    data_;  
    size_t   size_, capacity_; 
};
```

Invariants (must always be true for String s to be valid):

1. s.capacity_ == 0 if and only if s.data_ == nullptr
2. If s.capacity_ > 0 then:
   2.1 s.data_ points to a unique, free store–allocated array of s.capacity_ chars
   2.2 s.size_ + 1 <= s.capacity_
   2.3 s.data_[0], ..., s.data_[s.size_ – 1] are initialized
Our own String type

This is C++:

```cpp
struct String
{
    char* data_;    
    size_t size_, capacity_; 
};
```

Invariants (must always be true for String s to be valid):

1. `s.capacity_ == 0` if and only if `s.data_ == nullptr`
2. If `s.capacity_ > 0` then:
   2.1 `s.data_` points to a unique, free store–allocated array of `s.capacity_` chars
   2.2 `s.size_ + 1 <= s.capacity_`
   2.3 `s.data_[0], ..., s.data_[s.size_ - 1]` are initialized
   2.4 `s.data_[s.size_] == '\0'`
Some C++ stuff from the previous slide

- `struct` declarations create a type, so we can use `String` as a type, not just `struct String`
- The null pointer is named `nullptr` instead of `NULL`
- The C++ version of the heap is called the `free store` (and we will manage it using `new` and `delete` instead of `malloc` and `free`
Some C++ stuff from the previous slide

- `struct` declarations create a type, so we can use `String` as a type, not just `struct String`
- The null pointer is named `nullptr` instead of `NULL`
- The C++ version of the heap is called the *free store* (and we will manage it using `new` and `delete` instead of `malloc` and `free`)

(The old C ways still work, but we won’t use them in C++. For example, we will see later why `nullptr` is better than `NULL`.)

The String type lifecycle

The invariant says that the data_ member variable points to an object that is *unique*—meaning that no other String’s data_ points to the same object.

Implications:
The String type lifecycle

The invariant says that the data_ member variable points to an object that is *unique*—meaning that no other String’s data_ points to the same object.

Implications:

1. To create a (non-empty) String, we need to allocate a new object for data_ to point to.
The String type lifecycle

The invariant says that the data_ member variable points to an object that is unique—meaning that no other String’s data_ points to the same object.

Implications:

1. To create a (non-empty) String, we need to allocate a new object for data_ to point to.
2. To copy-assign one String object to another, we need to copy the contents of data_, not the pointer data_ itself.
The String type lifecycle

The invariant says that the data_ member variable points to an object that is *unique*—meaning that no other String’s data_ points to the same object.

Implications:

1. To create a (non-empty) String, we need to allocate a new object for data_ to point to.
2. To copy-assign one String object to another, we need to copy the contents of data_, not the pointer data_ itself.
3. We need to delete (C++’s free) data_ each time we are done with a String object
The String type lifecycle

The invariant says that the data_ member variable points to an object that is *unique*—meaning that no other String’s data_ points to the same object.

Implications:

1. To create a (non-empty) String, we need to allocate a new object for data_ to point to.
2. To copy-assign one String object to another, we need to copy the contents of data_, not the pointer data_ itself.
3. We need to delete (C++’s free) data_ each time we are done with a String object.

C++ can do the above automatically for us, but we’ll do it manually first.
Faking it
Special functions

C++ manages the lifecycle of an object with three kinds of special functions:

- Constructors initialize an uninitialized object
- Assignment operators copy or move from one initialized object to another
- The destructor frees an object’s resources
Special functions

C++ manages the lifecycle of an object with three kinds of special functions:

- Constructors initialize an uninitialized object
- Assignment operators copy or move from one initialized object to another
- The destructor frees an object’s resources

Our process for faking it:

1. Define an object
2. Call one constructor, once
3. Use the object for whatever
4. Call the destructor once
5. Don’t use the object again after that
Object lifecycle state diagram

- uninitialized
- valid
- defunct

Operations:
- constructor
- destructor

States:
- allocation
- deallocation
C++ automatically constructs after allocation and destroys before deallocation, but first we’re going to do it ourselves.
DIY String constructors (1/2)

// Constructs an empty string:
void String_construct_default(String*);

// Constructs by copying another String:
void String_construct_copy(String*,
const String* other);

// Constructs by moving (stealing the resources
// of) another String:
void String_construct_move(String*,
String* other);
DIY String constructors (1/2)

// Constructs an empty string:
void String_construct_default(String*);

// Constructs by copying another String:
void String_construct_copy(String*,
    const String* other);

// Constructs by moving (stealing the resources
// of) another String:
void String_construct_move(String*,
    String* other);

In C++ these three constructors will be special, in the sense
that it uses them in particular places; for example, it uses the
copy constructor every time you initialize a String from
another String, which includes passing or returning by value.
We also want constructors that are specific to our String type:

```c
// Constructs from a C string:
void String_construct_c_str(String*,
                             const char* s);
```

```c
// Constructs from the range [begin, end):
void String_construct_range(String*,
                             const char* begin,
                             const char* end);
```
DIY String destructor

// Frees any resources owned this `String`.
void String_destroy(String*);

Using our DIY constructors and destructor

const char* c_str = "hello\0world";
String s1, s2, s3, s4;

// Initialize s1 to the empty string:
String_construct_default(&s1);

// Initialize s2 to the string "hello":
String_construct_c_str(&s2, c_str);

// Initialize s3 to the string "hello\0world":
String_construct_range(&s3, c_str, c_str + 11);

// Initialize s4 to be a copy of s3:
String_construct_copy(&s4, &s3);

String_destroy(&s1); String_destroy(&s2);
String_destroy(&s3); String_destroy(&s4);
Using our DIY constructors and destructor

const char* c_str = "hello\0world";
String s1, s2, s3, s4;

// Initialize s1 to the empty string:
String_construct_default(&s1);

// Initialize s2 to the string "hello":
String_construct_c_str(&s2, c_str);

// Initialize s3 to the string "hello\0world":
String_construct_range(&s3, c_str, c_str + 11)

// Initialize s4 to be a copy of s3:
String_construct_copy(&s4, &s3);

String_destroy(&s1); String_destroy(&s2);
String_destroy(&s3); String_destroy(&s4);
const char* c_str = "hello\0world";
String s1, s2, s3, s4;

// Initialize s1 to the empty string:
String_construct_default(&s1);

// Initialize s2 to the string "hello":
String_construct_c_str(&s2, c_str);

// Initialize s3 to the string "hello\0world":
String_construct_range(&s3, c_str, c_str + 11)
Using our DIY constructors and destructor

const char* c_str = "hello\0world";
String s1, s2, s3, s4;

// Initialize s1 to the empty string:
String_construct_default(&s1);

// Initialize s2 to the string "hello":
String_construct_c_str(&s2, c_str);

// Initialize s3 to the string "hello\0world":
String_construct_range(&s3, c_str, c_str + 11)

// Initialize s4 to be a copy of s3:
String_construct_copy(&s4, &s3);
Using our DIY constructors and destructor

const char* c_str = "hello\0world";
String s1, s2, s3, s4;

// Initialize s1 to the empty string:
String_construct_default(&s1);

// Initialize s2 to the string "hello":
String_construct_c_str(&s2, c_str);

// Initialize s3 to the string "hello\0world":
String_construct_range(&s3, c_str, c_str + 11)

// Initialize s4 to be a copy of s3:
String_construct_copy(&s4, &s3);

String_destroy(&s1); String_destroy(&s2);
String_destroy(&s3); String_destroy(&s4);
void String_construct_default(String* this)
{
    this->capacity_ = 0;
    this->size_    = 0;
    this->data_    = nullptr;
}
// Never do this:
#define this actually_not_this

void String_construct_default(String* this) {
    this->capacity_ = 0;
    this->size_ = 0;
    this->data_ = nullptr;
}
void String_construct_move(String* this,
                        String* other) {
    this->capacity_ = other->capacity_;
    this->size_ = other->size_;  
    this->data_ = other->data_;  

    other->capacity_ = 0;
    other->size_ = 0;
    other->data_ = nullptr;
}
DIY constructor implementations (3/5)

```
void String_construct_copy(String* this,
                          const String* other)
{
    String_construct_range(
        this,
        other->data_,
        other->data_ + other->size_);
}
```
void String_construct_c_str(String* this, const char* s) {
    String_construct_range(this, s, s + std::strlen(s));
}
DIY constructor implementations (5/5)

```cpp
void String_construct_range(String* this,
   const char* begin,
   const char* end)
{
    size_t size = end - begin;

    if (size == 0) {
        String_construct_default(this);
        return;
    }

    this->capacity_ = size + 1;
    this->size_ = size;
    this->data_ = new char[size + 1];
    this->data_[size] = '\0';
    std::memcpy(this->data_, begin, size);
}
```
Okay, so new and delete

The **new** operator allocates on the free store, and the **delete** operator deallocates **new**-allocated objects.

(What’s the free store? Just like the heap, but a different place.)
Okay, so `new` and `delete`

The `new` operator allocates on the free store, and the `delete` operator deallocates `new`-allocated objects.

(What’s the free store? Just like the heap, but a different place.)

Each comes in two basic forms:

<table>
<thead>
<tr>
<th>allocate</th>
<th>deallocate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single object:</td>
<td><code>T* p = new T;</code></td>
</tr>
<tr>
<td>Array:</td>
<td><code>T* p = new T[N];</code></td>
</tr>
</tbody>
</table>

(UB if your `delete` form doesn’t match the `new` form.)
How does new differ from malloc?

It never returns nullptr and always calls constructors:

```cpp
T* operator new()
{
    T* result = free_store_malloc(sizeof(T));
    if (! result) throw something;
    T_construct_default(result);
    return result;
}
```

For symmetry, delete calls destructors:

```cpp
void operator delete(T* ptr)
{
    T_destroy(ptr);
    free_store_free(ptr);
}
```
How does new differ from malloc?

It never returns nullptr and always calls constructors:

```cpp
T* operator new()
{
    T* result = free_store_malloc(sizeof(T));
    if (! result) throw something;
    T_construct_default(result);
    return result;
}
```

For symmetry, delete calls destructors:

```cpp
void operator delete(T* ptr)
{
    T_destroy(ptr);
    free_store_free(ptr);
}
```
How new[] might work

```c
struct layout {
    size_t size;
    T data[0];
};

T* operator new[](size_t size) {
    layout* result = free_store_malloc(sizeof(layout) + size * sizeof(T));
    if (!result) throw something;
    result->size = size;
    for (size_t i = 0; i < size; ++i)
        T_construct_default(result->data + i);
    return result->data;
}
```
Implementing the destructor

```cpp
void String_destroy(String* this) {
    delete [] this->data_;  
}
```
Unlike constructors, assigners require that this already be initialized.

// Makes `this` a copy of `other`:
void String_assign_copy(String* this,
    const String* other)

// Moves contents of `other` to `this`,
// leaving `other` empty:
void String_assign_move(String* this,
    String* other)
void String_assign_move(String* this, String* other)
{
    delete [] this->data_;  

    this->capacity_ = other->capacity_;  
    this->size_ = other->size_;  
    this->data_ = other->data_;  

    other->capacity_ = 0;  
    other->size_ = 0;  
    other->data_ = nullptr;  
}
Implementing the assigners (2/2)

```c
void String_assign_copy(String* this,
                        const String* other)
{
    // Reallocate only if capacity is insufficient:
    if (other->size_ > 0 &&
        other->size_ + 1 > this->capacity_) {
        char* new_data = new char[other->size_ + 1];
        delete [] this->data_;
        this->data_ = new_data;
        this->capacity_ = other->size_ + 1;
    }

    if (this->data_ && other->data)
        std::memcpy(this->data_, other->data_,
                    other->size_ + 1);
    else if (this->data_) this->data_[0] = '\0';

    this->size_ = other->size_;}
```
Non-lifecycle operations

```c
bool String_empty(const String* this);
size_t String_size(const String* this);
char String_index(const String* this, size_t index);
char* String_index_mut(String* this, size_t index);
void String_push_back(String* this, char c);
void String_pop_back(String* this);
```
void String_push_back(String* this, char c) {
    ensure_capacity(this, this->size_ + 2);
    this->data_[this->size_++] = c;
    this->data_[this->size_] = '\0';
}

void String_pop_back(String* this) {
    this->data_[--this->size_] = '\0';
}
An important helper

```cpp
static void ensure_capacity(String* this,
                           size_t min_cap)
{
    if (this->capacity_ < min_cap) {
        size_t new_cap =
            std::max(min_cap, 2 * this->capacity_);
        char* new_data = new char[new_cap];
        if (this->data_)
            std::memcpy(new_data, this->data_,
                        this->size_ + 1);
        delete [] this->data_;            
        this->data_ = new_data;          
        this->capacity_ = new_cap;
    }
}
```
Doing it for real
A borrowed string type

We can use this to represent borrowed strings:

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

const char* s = "hello\0world";
range r1 {s, s + std::strlen(s)};
assert(r1.end - r1.begin == 5);
range r2 {s, s + 11};
assert(r2.end - r2.begin == 11);
A borrowed string type

We can use this to represent borrowed strings:

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

const char* s = "hello\0world";
range r1 {s, s + std::strlen(s)};
assert(r1.end - r1.begin == 5);
```
A borrowed string type

We can use this to represent borrowed strings:

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

const char* s = "hello\0world";
range r1 {s, s + std::strlen(s)};
assert(r1.end - r1.begin == 5);

range r2 {s, s + 11};
assert(r2.end - r2.begin == 11);
```
Adding a member function

In C++, struct members are not only variables—they can be functions (and types) as well. Here we add one member function for determining the size of a range:

```cpp
struct range
{
    size_t size() const;

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

```cpp
c = "hello\0world"
range r3 {s, s + 11};
assert(r3.size() == 11);
```
Adding a member function

In C++, `struct` members are not only variables—they can be functions (and types) as well. Here we add one member function for determining the size of a range:

```cpp
struct range
{
    size_t size() const;

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

const char* s = "hello\0world";
range r3 {s, s + 11};
assert(r3.size() == 11);
```
Why a member function?

Why not this?:

```cpp
template<size_t size(range r) {
    return r.end - r.begin;
}
```
Why a member function?

Why not this?:

```c
size_t size(range r) {
  return r.end - r.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`

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

Also, `this->` is implicit on member names!
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:
- `range::size` names the `size` member function of the `range` type in general
- `a_range.size` means the `size` member function on a particular instance of `range` (`a_range`)
- `a_range.begin` means the `begin` member variable of a particular instance of `range` (`a_range`)
- `range::begin` (usually) doesn't mean anything
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:

- range::size names the size member function of the range type in general
- a_range.size means the size member function on a particular instance of range (a_range)
- a_range.begin means the begin member variable of a particular instance of range (a_range)
- range::begin (usually) doesn’t mean anything
Operator overloading

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

// Declaration (in .h):
bool operator==(range, range);

// Definition (in .cpp):
#include <algorithm>

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

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

const char* s = "hello";
range r1 {s, s + 5};  // error: no such constructor
range r2 {s, 5};      // all good
```
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:

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

```cpp
const char* s = "hello";
range r1 {s, s + 5};  // error: no such constructor
range r2 {s, 5};      // all good
```
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:

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

const char* s = "hello";
range r1 {s, s + 5}; // error: no such constructor
range r2 {s, 5};    // all good
```
How does that make it more convenient though?

Multiple constructors, chosen by argument type:

```c
struct range
{
    range(const char* begin, const char* end);
    range(const char* start, size_t size);
    range(const char* c_str);
    range(const String* s);
    ...
};
```
How does that make it more convenient though?

Multiple constructors, chosen by argument type:

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

```c
const char* s1 = "hello\0world";
String s3(s1, s1 + 11);
```
How does that make it more convenient though?

Multiple constructors, chosen by argument type:

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

```c
const char* s1 = "hello\0world";
String s3(s1, s1 + 11); // future String constructor
range r1(s1, s1 + 11); // 1st constructor
range r2(s1, 11); // 2nd constructor
range r3(s1); // 3rd constructor
range r4(&s2); // 4th constructor
```
Defining constructors

Constructors have a special syntax for initializing member variables:

```cpp
range::range(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:

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

Constructors can also delegate to other constructors:

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

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

```cpp
range::range(const String* s)
    : range(s->c_str(), s->size()) {}
```
Constructors can enforce invariants

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

C++ can help us guarantee this for all ranges.
Constructors can enforce invariants

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

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

The first step is to avoid constructing invalid ranges.
Constructors can enforce invariants

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

C++ can help us guarantee this for all ranges. The first step is to avoid constructing invalid ranges. We could fix improper ranges:

```cpp
range::range(const char* begin0, const char* end0)
    : begin{begin0},
      end{end0 < begin0 ? begin0 : end0}
{ }
```
Constructors can enforce invariants

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

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

The first step is to avoid constructing invalid ranges. Or we could error on improper ranges:

```cpp
range::range(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 range.
Okay, but what if I…?
Okay, but what if I…?

```cpp
const char* s = "hello";
range r1(s);
```
Okay, but what if I...?

```c
const char* s = "hello";
range r1(s);
s.end = s.begin - 3;
```
Okay, but what if I...?

const char* s = "hello";
range r1(s);
s.end = s.begin - 3;

Oh no!
New idea: Access modifiers

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

```cpp
struct Name {
    // visible to all

private:
    // visible only to other members

public:
    // visible to all
}
```

• ```cpp
class T { ... };
```

• ```cpp
struct T { ... };
```

• ```cpp
class T { public: ... };
```
New idea: Access modifiers

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

```cpp
struct Name
{
    // visible to all

private:
    // visible only to other members

public:
    // visible to all
};
```

- `class T { ... }; ≡ struct T { private: ... };`
- `struct T { ... }; ≡ class T { public: ... };`
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 range class

class range
{
public:

    // Constructors:
    range(const char*, const char*);
    range(const String*);
    ...  

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

private:

    const char *begin_, *end_;  
};
Implementing the accessors

```c++
const char* range::begin() const
{
    return begin_;
}

const char* range::end() const
{
    return end_;  
}
```
Non-member functions must use accessors

Doesn’t work because range::begin_ and range::end_ are private:

```cpp
bool operator==(range a, range b) {
    return a.size() == b.size() &&
    std::equal(a.begin_, a.end_, b.begin_);
}
```
Non-member functions must use accessors

Works because `range::begin()` const and `range::end()` const are public:

```cpp
bool operator==(range a, range b)
{
    return a.size() == b.size() &&
           std::equal(a.begin(), a.end(), b.begin());
}
```
Non-member functions must use accessors

Works because `range::begin()` `const` and `range::end()` `const` are public:

```cpp
bool operator==(range a, range 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

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
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:

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

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

 private:
   const char* start_;  
   size_t size_;  
};
```
– Next time: Real RAII, really? —