The C++ Object Lifecycle

EECS 211

Winter 2019
Initial code setup

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

- Owned string type concept
- Faking it
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_;  // data_ points to the string data (array of characters)
    size_t size_, capacity_;  // size_ is the actual number of characters in our string
};
```

The idea:

- `data_` points to the string data (array of characters)
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)
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_;  // 1. if data_ is null, capacity_ is 0
    size_t size_, capacity_;  // 2. size_ + 1 <= 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. \( s.\text{capacity}_{} == 0 \) if and only if \( s.\text{data}_{} == \text{nullptr} \)
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_char
   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
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

operations

constructor

defunct

destructor

allocation

deallocation

C++ automatically constructs after allocation and destroys before deallocation, but first we're going to do it ourselves.
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);

// 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);
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)

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

```c
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

```c
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;              
}
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));
}
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>Single object:</th>
<th>allocate</th>
<th>deallocate</th>
</tr>
</thead>
<tbody>
<tr>
<td>T* p = new T;</td>
<td>delete p;</td>
<td></td>
</tr>
<tr>
<td>T* p = new T[N];</td>
<td>delete [] p;</td>
<td></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

```c
void String_destroy(String* this)
{
    delete [] this->data_;  
}
```
DIY “Assigners”

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)
Implementing the assigners (1/2)

```c++
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)

```cpp
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

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;
    }
}
```
– After the exam: Intro to GE211 —