Resource Acquisition Is Initialization

CS 211

Road map

Resource Acquisition Is Initialization

The problem: resource leaks

Road map

Resource Acquisition Is Initialization

The problem: resource leaks

The C++ solution: destructors

Road map

Resource Acquisition Is Initialization

The problem: resource leaks

The C++ solution: destructors

Example: an `Owned_string` class

Up next

Resource Acquisition Is Initialization

The problem: resource leaks

The C++ solution: destructors

Example: an `Owned_string` class

What is a resource?

Abstractly:

- Something you need to get your computation done
- that you can run out of,
- so you need to keep track of what you're using and release what you aren't.

Examples of resources

- memory, of course!
- file handles
- network sockets
- database sessions & transactions
- an acquired *lock* (in concurrent programming)

How resources leak

```
#include <cstdio>

void handle_file(std::string const& name)
{
    FILE *f = fopen(name.c_str(), "r");
    ⋮


    ⋮
    fclose(f);
}
```

How resources leak

```
#include <cstdio>

void handle_file(std::string const& name)
{
    FILE *f = fopen(name.c_str(), "r");
    ⋮
    if (something_came_up) return;
    ⋮
    fclose(f);
}
```

How resources leak

```
#include <cstdio>

void handle_file(std::string const& name)
{
    FILE *f = fopen(name.c_str(), "r");
    ⋮
    if (something_came_up) return;
    ⋮
    fclose(f);
}
```

When you have multiple resources—maybe several different kinds—this can get quite complicated

Now add exceptions

Non-local control makes things worse:

```
void helper()
{
    if (we_have_a_problem) throw std::runtime_error("Oops");
    ⋮
}

void handle_file(std::string const& name)
{
    FILE *f = fopen(name.c_str(), "r");
    ⋮
    helper();  // might never even return!
    ⋮
    fclose(f);
}
```

Up next

Resource Acquisition Is Initialization

The problem: resource leaks

The C++ solution: destructors

Example: an `Owned_string` class

Destructors are guaranteed to run

C++ provides a class `std::ifstream` for reading from a file; its destructor, `std::ifstream::~ifstream()`, closes the file.

```
#include <fstream>

void handle_file(std::string const& name)
{
    std::ifstream f(name, "r");
    ⋮




    ⋮
} // f.~ifstream() happens automatically here
```

Destructors are guaranteed to run

C++ provides a class `std::ifstream` for reading from a file; its destructor, `std::ifstream::~ifstream()`, closes the file.

```
#include <fstream>

void handle_file(std::string const& name)
{
    std::ifstream f(name, "r");
    ⋮
    if (something_came_up) return;
    ⋮


    ⋮
} // f.~ifstream() happens automatically here
```

Destructors are guaranteed to run

C++ provides a class `std::ifstream` for reading from a file; its destructor, `std::ifstream::~ifstream()`, closes the file.

```
#include <fstream>

void handle_file(std::string const& name)
{
    std::ifstream f(name, "r");
    ⋮
    if (something_came_up) return;
    ⋮
    might_throw();  // might not return, but that's okay!
    ⋮
} // f.~ifstream() happens automatically here
```

How to declare a destructor

A destructor is a member whose name is ~ followed by the name of the class or struct, with no result type and no arguments:

```
class My_class
{
    ⋮

public:
    ~My_class();

    ⋮
};
```

Minimal destructor example

```
struct Noisy
{
    explicit Noisy(std::string const& s);
    ~Noisy();
private:
    std::string who_;
};
```

Minimal destructor example

```
struct Noisy
{
    explicit Noisy(std::string const& s);
    ~Noisy();
private:
    std::string who_;
};

Noisy::Noisy(std::string const& s) : who_(s) { }

Noisy::~Noisy()
{
    std::cerr << who_ << " waves\n";
}
```

Using `Noisy`

```
void f()
{
    Noisy alice("Alice");
    Noisy bob("Bob");




}
```

Using Noisy

```
void f()
{
    Noisy alice("Alice");
    Noisy bob("Bob");









}  // prints "Bob waves\nAlice waves\n"
```

Using Noisy

```
void f()
{
    Noisy alice("Alice");
    Noisy bob("Bob");

    {
        Noisy carol("Carol");
    }



}  // prints "Bob waves\nAlice waves\n"
```

Using Noisy

```
void f()
{
    Noisy alice("Alice");
    Noisy bob("Bob");

    {
        Noisy carol("Carol");
    } // prints "Carol waves\n"



}  // prints "Bob waves\nAlice waves\n"
```

Using Noisy

```
void g(Noisy n)
{ }

void f()
{
    Noisy alice("Alice");
    Noisy bob("Bob");

    {
        Noisy carol("Carol");
    } // prints "Carol waves\n"

    g(alice);

}  // prints "Bob waves\nAlice waves\n"
```

Using Noisy

```
void g(Noisy n)
{ }

void f()
{
    Noisy alice("Alice");
    Noisy bob("Bob");

    {
        Noisy carol("Carol");
    } // prints "Carol waves\n"

    g(alice); // prints "Alice waves\n"

}  // prints "Bob waves\nAlice waves\n"
```

Up next

Resource Acquisition Is Initialization

The problem: resource leaks

The C++ solution: destructors

Example: an `Owned_string` class

How does `std::string` work?

- Constructors allocate a free-store* `char` array
- Destructor deallocates the array

* The free store C++'s version of the heap

Starting our `Owned_string` class

```
class Owned_string
{
public:
    Owned_string();
    Owned_string(char const *begin, char const *end);
    Owned_string(string_view);

    ~Owned_string();

    ⋮

private:
    std::size_t size_;      // logical size of string
    std::size_t capacity_;  // allocated size of 'data_'
    char       *data_;      // ptr to char array (or null)
};
```

What is `string_view`?

A type that borrows the data of a string:

```
class string_view
{
public:
    string_view();
    string_view(char const *begin, char const *end);
    string_view(char const *c_str);

    size_t size() const;
    bool empty() const;

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

    ⋮
};
```

Destructor implementation

`Owned_string` owns its data, so when it gets destroyed it deletes the data:

```
Owned_string::~Owned_string()
{
    delete [] data_;
}
```

Default constructor implementation

We represent the empty string using *nullptr* for the data:

```
Owned_string::Owned_string()
        : size_     (0)
        , capacity_ (0)
        , data_     (nullptr)
{ }
```

Constructing from a string view

To construct from a string view, we allocate (unless empty) and then copy:

```
Owned_string::Owned_string(string_view sv)
        : size_     (sv.size())
        , capacity_ (size_)
        , data_     (capacity_ ? new char[capacity_]
                               : nullptr)
{
    if (data_) {
        std::copy(sv.begin(), sv.end(), data_);
    }
}
```

Constructing from a range

To construct from a range, we delegate to the string view constructor:

```
Owned_string::Owned_string(char const *begin,
                           char const *end)
        : Owned_string(string_view(begin, end))
{ }
```

Copying

Okay, so what does this do?:

```
void g()
{
    Owned_string s1("hello!");
    Owned_string s2 = s1;

}
```

Copying

Okay, so what does this do?:

```
void f(Owned_string s);

void g()
{
    Owned_string s1("hello!");
    Owned_string s2 = s1;
    f(s1);
}
```

C++ copies objects using a copy constructor

If you don't define one, this is what you get:

```
Owned_string::Owned_string(Owned_string const& that)
        : size_     (that.size_)
        , capacity_ (that.capacity_)
        , data_     (that.data_)
{ }
```

C++ copies objects using a copy constructor

If you don't define one, this is what you get:

```
Owned_string::Owned_string(Owned_string const& that)
        : size_     (that.size_)
        , capacity_ (that.capacity_)
        , data_     (that.data_)
{ }
```

So:

```
{
    Owned_string s1("hello!");
    Owned_string s2 = s1;

} // double delete here
```

C++ copies objects using a copy constructor

If you don't define one, this is what you get:

```
Owned_string::Owned_string(Owned_string const& that)
        : size_     (that.size_)
        , capacity_ (that.capacity_)
        , data_     (that.data_)
{ }
```

So:

```
{
    Owned_string s1("hello!");
    Owned_string s2 = s1;
    f(s1); // another delete here
} // double delete here
```

Defining our own copy constructor

```
class Owned_string
{
public:
    ⋮
    Owned_string(Owned_string const&); // special!
    ⋮
};
```

Defining our own copy constructor

```
class Owned_string
{
public:
    ⋮
    Owned_string(Owned_string const&); // special!
    ⋮
};

Owned_string::Owned_string(Owned_string const& that)
        : Owned_string(that.data_,
                       that.data_ + that.size_)
{ }
```

Copy construction $\neq$ assignment

```
void f()
{
    Owned_string s1("hello");
    Owned_string s2("world");

    // s3 starts uninitialized here:
    Owned_string s3 = s1;

    // s2 starts initialized here:
    s2 = s1;
}
```

Copy construction ≠ assignment

```
void f()
{
    Owned_string s1("hello");
    Owned_string s2("world");

    // s3 starts uninitialized here:
    Owned_string s3 = s1;

    // s2 starts initialized here:
    s2 = s1;
}
```

So to avoid leaks and double-frees, we also need to overload assignment!

The copy assignment operator, declared

```
class Owned_string
{
public:
    ⋮
    Owned_string& operator=(Owned_string const&);
    ⋮
};
```

The copy assignment operator, defined

```
Owned_string&
Owned_string::operator=(Owned_string const& that)
{
    if (that.size_ > capacity_) {
        delete [] data_;
        data_     = new char[that.size_];
        capacity_ = that.size_;
    }

    if (data_ && that.data_) {
        std::copy(that.begin(), that.end(), data_);
    }

    size_ = that.size_;
    return *this;
}
```

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;
    a = b;
    b = temp;
}
```

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;  // 1
    a = b;
    b = temp;
}
```

[1] allocates

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;  // 1
    a = b;                  // 2
    b = temp;
}
```

[1] allocates
[2] allocates if `a` is shorter

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;  // 1
    a = b;                  // 2
    b = temp;               // 3
}
```

[1] allocates
[2] allocates if `a` is shorter
[3] allocates if `b` is shorter

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;  // 1, 4
    a = b;                  // 2
    b = temp;               // 3
}
```

[1] allocates
[2] allocates if `a` is shorter
[3] allocates if `b` is shorter
[4] copies `char` data of `a`

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;  // 1, 4
    a = b;                  // 2, 5
    b = temp;               // 3
}
```

[1] allocates
[2] allocates if `a` is shorter
[3] allocates if `b` is shorter
[4] copies `char` data of `a`
[5] copies `char` data of `b`

Why overload `swap`?

The default `std::swap` will do this for `Owned_string`:

```
void std::swap(Owned_string& a, Owned_string& b)
{
    Owned_string temp = a;  // 1, 4
    a = b;                  // 2, 5
    b = temp;               // 3, 4
}
```

[1] allocates
[2] allocates if `a` is shorter
[3] allocates if `b` is shorter
[4] copies `char` data of `a` (twice)
[5] copies `char` data of `b`

Overloading `swap` for efficiency

```
class Owned_string
{
public:
    void swap(Owned_string& that) noexcept
    {
        std::swap(size_,     that.size_);
        std::swap(capacity_, that.capacity_);
        std::swap(data_,     that.data_);
    }
    ⋮
};

void swap(Owned_string& a, Owned_string& b)
{ a.swap(b); }
```

Overloading `swap` for efficiency

```
class Owned_string
{
public:
    void swap(Owned_string& that) noexcept
    {
        std::swap(size_,     that.size_);
        std::swap(capacity_, that.capacity_);
        std::swap(data_,     that.data_);
    } // ^ just swaps pointers
    ⋮
};

void swap(Owned_string& a, Owned_string& b)
{ a.swap(b); }
```

Overloading `swap` for efficiency

```
class Owned_string
{
public:
    void swap(Owned_string& that) noexcept
    {
        std::swap(size_,     that.size_);
        std::swap(capacity_, that.capacity_);
        std::swap(data_,     that.data_);
    } // ^ just swaps pointers (no allocation, no copy loops)
    ⋮
};

void swap(Owned_string& a, Owned_string& b)
{ a.swap(b); }
```

Toward string concatenation

```
class Owned_string
{
public:
    ⋮

    // Implicitly converts to `string_view`:
    operator string_view() const;

    // Grows capacity to accomodate `size() + additional`:
    void reserve(size_t additional);

    ⋮
};
```

Implementations

```
Owned_string::operator string_view() const
{
    if (data_)
        return string_view(data_, size_);
    else
        return string_view();
}

void Owned_string::reserve(size_t additional)
{
    if (size_ + additional)
        ensure_capacity_(size_ + additional);
}
```

String extension

```
class Owned_string
{
public:
    ⋮
    Owned_string& operator+=(string_view);
    ⋮
};

Owned_string& Owned_string::operator+=(string_view sv)
{
    reserve(sv.size());
    std::copy(sv.begin(), sv.end(), end());
    size_ += sv.size();
    return *this;
}
```

String concatenation

```
Owned_string
operator+(string_view a, string_view b)
{
    Owned_string result;
    result.reserve(a.size() + b.size());
    result += a;
    result += b;
    return result;
}

Owned_string
operator+(Owned_string&& a, string_view b)
{
    a += b;
    return a;
}
```

Move constructor

```
class Owned_string
{
public:
    Owned_string(Owned_string&&) noexcept;
    ⋮
}

Owned_string::Owned_string(Owned_string&& that)
        : size_(that.size_)
        , capacity_(that.capacity_)
        , data_(that.data_)
{
    that.capacity_ = that.size_ = 0;
    that.data_     = nullptr;
}
```

Move assignment operator

```
class Owned_string
{
public:
    ⋮
    Owned_string& operator=(Owned_string&&);
    ⋮
}

Owned_string&
Owned_string::operator=(Owned_string&& that)
{
    swap(that);
    return *this;
}
```