Resource Acquisition Is Initialization

CS 211

Winter 2020

Road map

- 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);
}
```

The C++ solution

Destructors are guaranteed to run

```
#include <fstream>

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



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

Destructors are guaranteed to run

```
#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

```
#include <fstream>

void handle_file(std::string const& name)
{
    std::ifstream f(name, "r");
    ...
    if (something_came_up) return;
    ...
    might_throw();  // might not return
    ...
} // 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();

    ...
};
```

Simple destructor example

```
class Noisy
{
public:
    Noisy(std::string const&);
    ~Noisy();
private:
    std::string name_;
};
```

Simple destructor example

```
class Noisy
{
public:
    Noisy(std::string const&);
    ~Noisy();
private:
    std::string name_;
};

Noisy::Noisy(std::string const& name)
        : name_(name) { }

Noisy::~Noisy()
{
    std::cerr << name_ << "␣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"
```

An owned string class

How does `std::string` work?

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

Starting our `Owned_string` class

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

    ~Owned_string();
    ...

private:
    size_t size_;     // logical size of string
    size_t capacity_; // allocated size of `data_`
    char*  data_;
};
```

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 a range we allocate (unless empty) and then copy:

```
Owned_string::Owned_string(string_view sv)
        : size_     {sv.size()}
        , capacity_ {size_ ? size + 1 : 0}
        , data_     {capacity_ ? new char[capacity_]
                               : nullptr}
{
    if (data_) {
        std::copy(sv.begin(), sv.end(), data_);
        data_[size_] = '\0';
    }
}
```

Constructing from a range

We can delegate to the string view constructor:

```
Owned_string::Owned_string(const char* begin,
                           const char* 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(Owned_string const&); // special!
    ...

    ~Owned_string();
    ...
};

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

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;
}
```

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_ && that.size_ + 1 >= capacity_) {
        delete [] data_;
        data_     = new char[that.size_ + 1];
        capacity_ = that.size_ + 1;
    }

    if (data_) {
        if (that.data_)
            std::copy(that.begin(), that.end() + 1, data_);
        else data_[0] = '\0';
    }

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

Overloading swap

```
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);
}
```

Toward string concatenation

```
class Owned_string
{
public:
    ...

    // Converts to a `string_view`:
    operator string_view() const;

    // Grows capacity:
    void reserve(size_t additional);

    ...
};
```

Implementations

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

void Owned_string::reserve(size_t additional)
{
    if (size_ + additional)
        ensure_capacity_(size_ + additional + 1);
}
```

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();
    *end() = '\0';
    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 std::move(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::operarator=(Owned_string&& that)
{
    swap(that);
    return *this;
}
```

– Next: Figuring Stuff Out —