RAII and Smart Pointers

Ali Malik
malikali@stanford.edu
Game Plan

- Recap
- Conversion Operators
- RAII
- Smart Pointers
Recap
Initialisation vs Assignment

Initialisation:

Transforms an object’s initial junk data into valid data.

Assignment:

Replaces existing valid data with other valid data.
Constructors

Normal Constructor:

- What you are used to!

Copy Constructor

- Initialise an instance of a type to be a copy of another instance

Copy Assignment

- Not a constructor
- Assign an instance of a type to be a copy of another instance
The Rule of Three

If you implement a copy constructor, assignment operator, or destructor, you should implement the others, as well
Conversion Operators

Let you define how a class can be converted to other types.

For example, we could define a conversion of the `MyVector` to a `bool` to be false if the vector is empty and true otherwise.
Conversion Operators

Converting to `Type` works by overloading the `Type()` operator. Doesn’t have a return value.

```cpp
class MyClass {

public:
    operator Type() {
        // return something of type Type
    }
}
```
Conversion Operators

An example defining a `bool` conversion for `MyVector`:

class MyVector {
public:
    operator bool() {
        return empty();
    }
};

MyVector v;
if(v) {
    cout << v[0] << endl;
}
Let’s first talk about the abstraction of a resource.

We will look at file opening and closing in C as a case study.
C File I/O

To read a file in C, you need to:

1. **Open** the file with `fopen`
2. **Read** data using `fgets`
3. **Close** the open file with `fclose`

If a programmer doesn’t remember to close an open file, bad things happen (memory leaks, crashes etc.)
C File I/O

You can think of this as a resource.

What is a resource?

● Anything that exists in limited supply.
● Something you have to acquire and release.

Examples: memory, open files, sockets etc.
C File I/O

To read a file in C, you need to:

1. Open the file with `fopen`
2. Read data using `fgets`
3. Close the open file with `fclose`

If a programmer doesn’t remember to close an open file, bad things happen (memory leaks, crashes etc.)
C File I/O

To read a file in C, you need to:

1. Open the file with `fopen` // acquire
2. Read data using `fgets`
3. Close the open file with `fclose`

If a programmer doesn’t remember to close an open file, bad things happen (memory leaks, crashes etc.)
C File I/O

To read a file in C, you need to:

1. Open the file with `fopen` // acquire
2. Read data using `fgets`
3. Close the open file with `fclose` // release

If a programmer doesn’t remember to close an open file, bad things happen (memory leaks, crashes etc.)
## Resources

Other examples of resources:

<table>
<thead>
<tr>
<th></th>
<th>Acquire</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Files</strong></td>
<td>fopen</td>
<td>fclose</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>new, new[]</td>
<td>delete, delete[]</td>
</tr>
<tr>
<td><strong>Locks</strong></td>
<td>lock, try_lock</td>
<td>unlock</td>
</tr>
<tr>
<td><strong>Sockets</strong></td>
<td>socket</td>
<td>close</td>
</tr>
</tbody>
</table>
RAII
Resource Acquisition Is Initialisation
RAII

A modern C++ idiom.

When you **initialize** an object, it should already have **acquired** any resources it needs (in the constructor).

When an object goes out of scope, it should **release** every resource it is using (using the destructor).
RAII

Key points:

● There should never be a half-ready or half-dead object.
● When an object is created, it should be in a ready state.
● When an object goes out of scope, it should release its resources.
Key points:

- There should never be a half-ready or half-dead object.
- When an object is created, it should be in a ready state.
- When an object goes out of scope, it should release its resources.

The user shouldn’t have to do anything more.
C File I/O

How does C File I/O violate RAII?

```c
void printFile(const char* name) {
    // acquire file resource
    FILE* f = fopen(name, "r");

    // print contents of f

    // release file resource
    fclose(f);
}
```
C File I/O

How does C File I/O violate RAII?

```c
void printFile(const char* name) {
    // acquire file resource
    FILE* f = fopen(name, "r");

    // print contents of f

    // release file resource
    fclose(f);
}
```
C File I/O

How does C File I/O violate RAII?

```c
void printFile(const char* name) {
    // acquire file resource
    FILE* f = fopen(name, "r");

    // print contents of f

    // release file resource?
}
```
C File I/O

How does C File I/O violate RAII?

```c
void printFile(const char* name) {
    // acquire file resource
    FILE* f = fopen(name, "r");

    // print contents of f

    // release file resource?
}
```

f goes out of scope, but doesn’t release its resources.
C File I/O

What would be an RAII friendly solution for C File I/O?
class FileObj {
public:
    FILE* ptr;
    FileObj(char* name)
        : ptr(fopen(name, "r")) {}}

    ~FileObj() {
        fclose(ptr);
    }
};

C File I/O + RAII
C File I/O

Our new printFile method would look like:

```c
void printFile(const char* name) {
    // initialization will acquire resources
    FileObj fobj(name);

    // print contents of f

    // FileObj destructor will release resources
}
```
In fact, you have already been using RAII!

For example:

- You can create an ifstream and it will open the file
- When the ifstream goes out of scope, its destructor closes the file.

Don’t actually need to call the .close() method.
RAII is a bad name for the concept.

"The best example of why I shouldn't be in marketing"

"I didn't have a good day when I named that"

Bjarne Stroustrup, still unhappy with the name RAII in 2012
RAII - An Aside

A better name is probably:

**Constructor Acquires, Destructor Releases**

or

**Scope Based Resource Management**
Smart Pointers
Smart Pointers

What is another thing that violates RAII?

Raw Pointers and heap allocation!
Smart Pointers

Calls to `new` acquire resource (memory).

Calls to `delete` release resource.

But this is not automatically done when the pointers go out of scope.
Smart Pointers

But this is not automatically done when the pointers go out of scope:

```cpp
void rawPtrFn() {
    // acquire memory resource
    Node* n = new Node;

    // manually release memory
    delete n;
}
```
Smart Pointers

But this is not automatically done when the pointers go out of scope:

```c++
void rawPtrFn() {
    // acquire memory resource
    Node* n = new Node;

    // manually release memory
    delete n;
}
```

If we forget this, we leak memory.
Smart Pointers

What would be an RAII solution to this?

Have a class that

- **Allocates** the memory when initialized
- **Frees** the memory when destructor is called
- Allows access to underlying pointer
Smart Pointers

Let’s plan and write this up:

Smart Pointers
(=RAIIPtr_unique.pro)
First we make a smart pointer

```cpp
SmartPtr<int> x;
```
Smart Pointers

We then make a copy of our smart pointer

```cpp
SmartPtr<int> x;
```

```cpp
resource
```

```cpp
SmartPtr<int> y;
```

```cpp
resource
```

Data (heap)
Smart Pointers

When y goes out of scope, it deletes the heap data
This leaves `x` pointing at deallocated data

```cpp
SmartPtr<int> x;
```

`SmartPtr` < `int` > `x`;
If we dereference x or its destructor calls delete, we crash
If we dereference x or its destructor calls delete, we crash
Smart Pointers

Have to be careful when copying an RAII object

Don’t want two objects thinking they both exclusively own a resource
C++ already has built-in smart pointers.

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
unique_ptr

Similar to what we wrote earlier

Uniquely own its resource and deletes it when the object is destroyed

Cannot be copied!

```cpp
{
    std::unique_ptr<int> p(new int);
    // Use p
}
// Freed!
```
Resource can be stored by any number of `shared_ptr`

Deleted when none of them point to it

```cpp
{
    std::shared_ptr<int> p1(new int);
    // Use p1
    {
        std::shared_ptr<int> p2 = p1;
        // Use p1 and p2
    }
    // Use p1
}
// Freed!
```
shared_ptr

How are these implemented?

Reference counting!

Store an int that keeps track of the number currently referencing that data

- Gets incremented in copy constructor/copy assignment
- Gets decremented in destructor or when overwritten with copy assignment

Frees the resource when reference count hits 0
Smart Pointers

See Course Reader pg. 351 onwards for details. Let’s plan and write this up:

Smart Pointers
(RAINT_ptr_shared.pro)
weak_ptr

Used to deal with circular references of shared_ptr

Read documentation to learn more!
Next Time

Final Topics