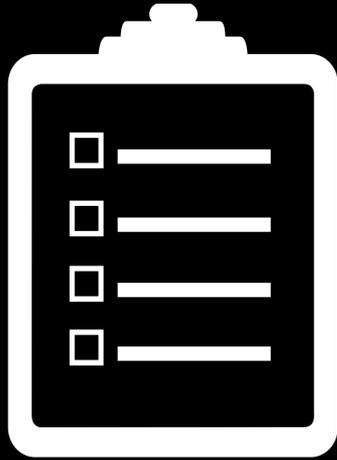


RAII and Smart Pointers

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

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.

```
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;
}
```

RAII

Resource

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:

	Acquire	Release
Files	<code>fopen</code>	<code>fclose</code>
Memory	<code>new, new[]</code>	<code>delete, delete[]</code>
Locks	<code>lock, try_lock</code>	<code>unlock</code>
Sockets	<code>socket</code>	<code>close</code>

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.

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.



The user shouldn't have to do anything more.

C File I/O

How does C File I/O violate RAI?

```
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 RAI?

```
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 RAI?

```
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 RAI?

```
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?

C File I/O + RAII

```
class FileObj {  
public:  
    FILE* ptr;  
    FileObj(char* name)  
        : ptr(fopen(name, "r")) {}  
  
    ~FileObj() {  
        fclose(ptr);  
    }  
};
```

C File I/O

Our new printFile method would look like:

```
void printFile(const char* name) {  
    // initialization will acquire resources  
    FileObj fobj(name);  
  
    // print contents of f  
  
    // FileObj destructor will release resources  
}
```

C File I/O

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 - An Aside

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 **A**cquires, Destructor **R**eleases

or

Scope **B**ased Resource **M**anagement

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:

```
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:

```
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

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

`RAIIPtr.pro`

Smart Pointers

C++ already has built-in smart pointers.

- `shared_ptr`
- `unique_ptr`
- `weak_ptr`

Next Time

Final Topics

