

# Resource Acquisition is Initialisation (RAII) Explained (/blog/software-design/resource-acquisition-is-initialisation-raii-explained/)

17 May, 2012 — Category: Software Design (/blog/category/software-design/)

In the competition to make the worst acronym, RAII (<http://en.wikipedia.org/wiki/RAII>) probably comes second after HATEOS (<http://en.wikipedia.org/wiki/HATEOAS>). Nevertheless, it is an important concept because it allows you to write safer code in C++ — a harsh, unforgiving language that is all too happy to help you shoot yourself in the foot.

This article will explain exception-safety and common pitfalls in C++. As we work out how to avoid these problems, we will accidentally discover RAII. Then, we will finish by defining exactly what RAII is, and where it is already being used.

## Exception-safety and Common Pitfalls in C++

The RAII paradigm is wide spread in modern C++, and for good reason. That reason is the presence of exceptions. C++ basically forces you to use exceptions, because there is no other way to signal the failure of a constructor method. You either ensure that all constructors never fail, which makes the design of your classes awkward, or you throw exceptions on failure.

Given that the use of exceptions is inevitable, your code needs to be exception-safe. Here are some common, yet unsafe, scenarios:

```
File f;
f.open("boo.txt");
//UNSAFE - an exception here means the file is never closed
loadFromFile(f);
f.close();

Dog* dog = new Daschund;
//UNSAFE - an exception here means the dog is never deleted
goToThePark(dog);
delete dog;

Lock* lock = getLock();
lock.aquire();
//UNSAFE - an exception here means the lock is never released
doSomething();
lock.release();
```

Not only will exceptions break your code, it's also easy to just forget to close a file. Maybe you close the file 99% of the time but there is one rare edge case that you forgot about. It would be nice if there was a safeguard for these situations — something that was guaranteed to close the file so we can't forget to do it.

## Finding A Solution

When we aquire a resource (such as opening a file) what we want is to guarantee that we run code to relinquish that resource (such as closing the file). Luckily, there is a way to guarantee that code will run in C++. **C++ guarantees that the destructors of objects on the stack will be called, even if an exception is thrown.**

So we just need to get the file closing code into the destructor of some class, and then make an instance of that class on the stack. Let's try it out with the file example:

```
class FileCloser {
public:
    FileCloser(File* file) {
        _file = file;
    }
    ~FileClose() {
        _file->close();
    }
private:
    File* _file;
}

// then we can use it like this
File f;
f.open("boo.txt");
FileCloser closer(&f);
//SAFE - Even if this throws an exception the FileCloser destructor will
//      run and close the file.
loadFromFile(f);
//don't need to close the file here because the FileCloser
//destructor will run at this point
```

This is a very naive solution with a couple of problems, so don't copy it, but it does give you an idea of what we are trying to achieve.

One problem with this is that we might forget to make a `FileCloser`, in the same way that we might forget to close the file. It would be much better if the `File` class could just close itself. Let's make a new class called `MyFile` that closes itself:

```
class MyFile {
public:
    MyFile() {}

    ~MyFile() {
        if(_file.isOpen()){
            _file.close();
        }
    }

    void open(const char* filename) {
        _file.open(filename);
    }

    bool isOpen() const {
        return _file.isOpen();
    }

    void close() {
        if(_file.isOpen()){
            _file.close();
        }
    }

    std::string readLine() {
        return _file.readLine();
    }

private:
    File _file;
};

// then we can use it like this
MyFile f;
f.open("boo.txt");
//SAFE - The MyFile destructor is guaranteed to run
loadFromFile(f);
```

That is looking a lot better. Using `MyFile` is simpler and cleaner than `FileCloser`, and it's exception-safe as well. We can still do better, though. What if someone calls the `open` method twice? Also, it's annoying to check `isOpen` everywhere.

Let's take a different approach. What if we make a class that represents an open file? That way, we

don't have to worry about opening twice, or closing twice, and we don't even have to check if the file is open.

```
class OpenFile {
public:
    OpenFile(const char* filename){
        //throws an exception on failure
        _file.open(filename);
    }

    ~OpenFile(){
        _file.close();
    }

    std::string readLine() {
        return _file.readLine();
    }

private:
    File _file;
};

// then we can use it like this
OpenFile f("boo.txt");
//exception safe, and no closing necessary
loadFromFile(f);
```

`OpenFile` is exception safe, and nice and simple. We have accidentally stumbled across RAII. Well it wasn't really accidental, but let's just pretend.

## What is RAII?

There are three parts to an RAII class:

- The resource is relinquished in the destructor (e.g. closing a file)
- Instances of the class are stack allocated
- The resource is aquired in the constructor (e.g. opening a file). This part is optional, but common.

RAII stands for "Resource Acquisition is Initialisation." The "resource acquisition" part of RAII is where you begin something that must be ended later, such as:

- Opening a file (and closing it later)
- Allocating some memory (and deallocating it later)
- Acquiring a lock (and releasing it later)

The "is initialisation" part means that the acquisition happens inside the constructor of a class. Want to open a file? Then the opening should happen in the constructor like we saw in the `OpenFile` example above. This isn't totally necessary, but I think it simplifies the classes. You could also argue that "initialisation" can happen outside of the constructor, shortly after the constructor is run.

Ironically, the acronym RAII doesn't explain the most important part of the design, which is that the relinquishing of the resource (closing, deallocating, etc.) must be put into the destructor.

The final part is to ensure that the instance is allocated on the *stack* and not on the *heap*. It is easy to see why if you compare the two:

```
std::string firstLineOf(const char* filename){
    OpenFile f("boo.txt"); //stack allocated
    return f.readLine();
    //File closed here. `f` goes out of scope and destructor is run.
}

std::string firstLineOf(const char* filename){
    OpenFile* f = new OpenFile("boo.txt"); //heap allocated
    return f->readLine();
    //DANGER WILL ROBINSON! Destructor is never run, because `f` is never
    //deleted
}
```

A better name for RAII would be something like "Scope-bound Resources," because you're tying the life of a resource to the life of a local variable, and the life of the local variable ends when it goes out of scope. Actually, let's change that to "Resources are bound irreversibly to scope," because then the acronym is RABITS. Everybody likes rabbits.

## Common Uses of RAII

When it comes to opening and closing files, `std::fstream` already has an RAII type of design because it closes itself in its destructor.

The Boost library uses RAII for locking and unlocking with classes like `boost::lock_guard` ([http://www.boost.org/doc/libs/1\\_49\\_0/doc/html/thread/synchronization.html#thread.synchronization.locks.lock\\_guard](http://www.boost.org/doc/libs/1_49_0/doc/html/thread/synchronization.html#thread.synchronization.locks.lock_guard)) and `boost::interprocess::scoped_lock`

([http://www.boost.org/doc/libs/1\\_49\\_0/doc/html/boost/interprocess/scoped\\_lock.html](http://www.boost.org/doc/libs/1_49_0/doc/html/boost/interprocess/scoped_lock.html)).

If you're writing modern C++ then no doubt you've heard of smart pointers. Smart pointers are RAII classes. They help avoid a whole bunch of problems, like forgetting to deallocate the memory, or deallocating it while it's still being used somewhere else.

`std::shared_ptr` is interesting, in that the "resource" is not the pointer itself — that's what `std::weak_ptr` does. Rather, the resource is a *guarantee* that the pointer is valid. Once the `shared_ptr` goes out of scope the pointer might still be valid, but you can't be sure because you've relinquished your guarantee. This just goes to show that a "resource" isn't always a physical thing like a file, or a peice of memory.

Enjoy this post?

Subscribe via RSS (/feed/)

Follow @tom\_dalling

## Comments

12 Comments Tom Dalling

 Login ▾ Recommend 7 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS Name **The** • 4 years ago

Thanks, this was a really clear explanation. I like the fact that you led us to the solution by going through other solutions to the same problem, as opposed to just dictating it right away

3 ^ | ▾ • Reply • Share &gt;

**Dima Molosnic** • 5 years ago

Finally! A good explanation of RAII.  
Thanks Tom. :)

1 ^ | ▾ • Reply • Share &gt;

**shabnam** • a month ago

thanks a ton for simple explanation.

^ | ▾ • Reply • Share &gt;

**HitokageProduction .** • a year ago

Thanks, so when dealing with heap allocations, the way to use RAII is to put them in a stack allocated instance of some wrapper class right?

^ | ▾ • Reply • Share &gt;

**Tom Dalling** Mod ➔ **HitokageProduction .** • a year ago

Yep. That's what smart pointers are, in a nutshell.

^ | ▾ • Reply • Share &gt;

**HitokageProduction .** ➔ **Tom Dalling** • a year ago

Thanks for the answer, good point. Have a nice day!

^ | ▾ • Reply • Share &gt;

**Mattias Johansson** • 2 years ago


Good introduction to RAII. In fact RAII is used throughout the C++ standard library for closing files, releasing mutexes, deleting allocated memory etc. See <http://en.cppreference.com/...> or <http://www.moderncplusplus...>

^ | ▾ • Reply • Share &gt;

**Crazy** • 2 years ago



## Subscribe

 [RSS \(/blog/feed/\)](/blog/feed/)

 [Twitter \(https://twitter.com/tom\\_dalling\)](https://twitter.com/tom_dalling)

## Recent Posts

The Pure Function As An Object (PFAAO) Ruby Pattern (</blog/ruby/pure-function-as-an-object-PFAAO-pattern/>)


FizzBuzz In Too Much Detail (</blog/software-design/fizzbuzz-in-too-much-detail/>)


Making Fruity Bat (a Flappy Bird clone) in Ruby (</blog/ruby/fruity-bat-flappy-bird-clone-in-ruby/>)


Modern OpenGL 08 – Even More Lighting: Directional Lights, Spotlights, & Multiple Lights (</blog/modern-opengl/08-even-more-lighting-directional-lights-spotlights-multiple-lights/>)


OpenGL in 2014 (</blog/modern-opengl/opengl-in-2014/>)


## Blog Categories


Cocoa (</blog/category/cocoa/>) ( 5  (</blog/category/cocoa/feed/>))

Coding Style/Conventions (</blog/category/coding-styleconventions/>) ( 3  (</blog/category/coding-styleconventions/feed/>))


Coding Tips (</blog/category/coding-tips/>) ( 4  (</blog/category/coding-tips/feed/>))


Miscellaneous (</blog/category/random-stuff/>) ( 1  (</blog/category/random-stuff/feed/>))

Modern OpenGL Series (</blog/category/modern-opengl/>) ( 10  (</blog/category/modern-opengl/feed/>))

Ruby (</blog/category/ruby/>) ( 2  (</blog/category/ruby/feed/>))

Software Design (</blog/category/software-design/>) ( 9  (</blog/category/software-design/feed/>))

Software Processes (</blog/category/software-processes/>) ( 1  (</blog/category/software-processes/feed/>))

Web (/blog/category/web/) ( 1  (/blog/category/web/feed/))

**Blog Archives**

February 2016 (/blog/2016/02/) ( 1 )

April 2015 (/blog/2015/04/) ( 1 )

February 2015 (/blog/2015/02/) ( 1 )

November 2014 (/blog/2014/11/) ( 1 )

September 2014 (/blog/2014/09/) ( 1 )

February 2014 (/blog/2014/02/) ( 1 )

April 2013 (/blog/2013/04/) ( 1 )

March 2013 (/blog/2013/03/) ( 1 )

February 2013 (/blog/2013/02/) ( 1 )

January 2013 (/blog/2013/01/) ( 2 )

December 2012 (/blog/2012/12/) ( 2 )

November 2012 (/blog/2012/11/) ( 2 )

July 2012 (/blog/2012/07/) ( 2 )

May 2012 (/blog/2012/05/) ( 2 )

March 2012 (/blog/2012/03/) ( 2 )

December 2011 (/blog/2011/12/) ( 1 )


December 2010 (/blog/2010/12/) ( 1 )


November 2010 (/blog/2010/11/) ( 1 )


May 2010 (/blog/2010/05/) ( 1 )


April 2010 (/blog/2010/04/) ( 1 )
February 2010 (/blog/2010/02/) ( 1 )
December 2009 (/blog/2009/12/) ( 1 )
November 2009 (/blog/2009/11/) ( 3 )
October 2009 (/blog/2009/10/) ( 2 )
July 2009 (/blog/2009/07/) ( 1 )
June 2009 (/blog/2009/06/) ( 1 )
May 2009 (/blog/2009/05/) ( 1 )

© 2009 – 2016 Tom Dalling

 ([https://twitter.com/tom\\_dalling](https://twitter.com/tom_dalling))

 (<https://github.com/tomdalling>)

 (<https://stackoverflow.com/users/108105/tom-dalling>)

 ([mailto:tom at tomdalling com](mailto:tom@tomdalling.com))