Back To Basics
RAII

ANDRE KOSTUR

Cppcon
The C++ Conference

2022
September 12th-16th

Agenda

- What is a “Resource”
- What issues do we have handling Resources
- RAII to manage resources
- Examples in the Standard
- Implementing a RAII class

Resources

A resource in C++ is some facility or concept that you gain access to by a statement or expression, and you release or dispose of that facility or concept by some other statement or expression.

Common Resources

Resource	Acquire	Dispose
Memory	`p = new T;`	`delete p;`
POSIX File	`fp = fopen( “filename”, “r”);`	`fclose(fp);`
Joinable threads	`pthread_create( &p, NULL, fn, NULL);`	`pthread_join(p, &retVal);`
Mutex locking	`pthread_mutex_lock(&mut);`	`pthread_mutex_unlock( &mut);`

Resource Usage Issues

- Leak
- Use-after-disposal
- Double-disposal

We'll use a mutex as the example resource.

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    BufferClass buffer;
    src.readIntoBuffer(buffer);
    buffer.display();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();   // Acquire a lock on the mutex
    BufferClass buffer;
    src.readIntoBuffer(buffer);
    buffer.display();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    BufferClass buffer;
    src.readIntoBuffer(buffer);
    buffer.display();
    return true;   // We didn’t unlock the mutex!
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    BufferClass buffer;
    src.readIntoBuffer(buffer);
    buffer.display();
    someMutex.unlock();   // Unlock the mutex
    return true;
}
```

Fixed. Now the mutex is correctly unlocked. But, we're not done.

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    BufferClass buffer;
    if (not src.readIntoBuffer(buffer)) {
        return false;
    }
    buffer.display();
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    BufferClass buffer;
    if (not src.readIntoBuffer(buffer)) {
        someMutex.unlock();
        return false;
    }
    buffer.display();
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    BufferClass buffer;
    if (not src.readIntoBuffer(buffer)) {  // Throws an exception!
        someMutex.unlock();
        return false;
    }
    buffer.display();
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) { // Throws an exception!
            someMutex.unlock();
            return false;
        }
        buffer.display();
    } catch (...) { someMutex.unlock(); throw; }
    someMutex.unlock();
    return true;
}
```

C++ Object Lifetimes

Objects have a defined beginning of life, and end of life. Both of those events have code which will automatically run: namely, constructors and destructors.

RAII

Resource Acquisition Is Initialization

In the purest sense of the term, this is the idiom where resource acquisition is done in the constructor of an “RAII class”, and resource disposal is done in the destructor of an “RAII class”.

Ownership

An RAII class is said to “own” the resource. It is responsible for cleaning up that resource at the appropriate time.

RAII Example: std::lock_guard

`std::lock_guard` is the standard RAII class to lock a single mutex during its construction, and unlock it during destruction.

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    someMutex.lock();
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) {
            someMutex.unlock();
            return false;
        }
        buffer.display();
    } catch (...) { someMutex.unlock(); throw; }
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    std::lock_guard lock{someMutex};   // Acquire
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) {
            someMutex.unlock();
            return false;
        }
        buffer.display();
    } catch (...) { someMutex.unlock(); throw; }
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    std::lock_guard lock{someMutex};
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) {
            someMutex.unlock();  // No longer necessary
            return false;
        }
        buffer.display();
    } catch (...) { someMutex.unlock(); throw; }
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    std::lock_guard lock{someMutex};
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) {
            return false;
        }
        buffer.display();
    } catch (...) { someMutex.unlock(); throw; }
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    std::lock_guard lock{someMutex};
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) {
            return false;
        }
        buffer.display();
    } catch (...) { throw; }
    someMutex.unlock();
    return true;
}
```

Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    std::lock_guard lock{someMutex};
    try {
        BufferClass buffer;
        if (not src.readIntoBuffer(buffer)) {
            return false;
        }
        buffer.display();
    } catch (...) { throw; }
    return true;
}
```

Final Sample Function

```
bool fn(std::mutex & someMutex, SomeDataSource & src) {
    std::lock_guard lock{someMutex};
    BufferClass buffer;
    if (not src.readIntoBuffer(buffer)) {
        return false;
    }
    buffer.display();
    return true;
}
```

Storage durations

So far we've only talked about automatic storage duration variables.

RAII works with any of the C++ object lifetimes.

```
void SomeClass::fn() {
   auto worker{std::jthread{[]{ /* do something */ }}};
   m_vec.push_back(std::move(worker));
}
```

RAII Example: std::unique_ptr

`std::unique_ptr` is the standard RAII class to hold a pointer, and dispose of it during destruction. By default, this corresponds to `new` and `delete`.

RAII Example: std::shared_ptr

The managed resource doesn’t even have to be a simple acquire/dispose pair.

`std::shared_ptr` represents a reference-counted shared pointer.

Other Standard RAII classes

Some other Standard RAII classes:

- `std::unique_lock`: a more sophisticated `std::lock_guard`, but you can unlock and relock it during its lifespan, and other more sophisticated things
- `std::jthread`: manages a joinable thread, and will automatically join to the thread during destruction
- `std::fstream`: opens and closes the files

Reclaim Responsibility

RAII classes may provide ways to get direct access to the enclosed resource.

RAII classes may even provide ways to break the resource out of the RAII class altogether.

Solves the problems?

- Leaks?
  - Yes. Automatic storage durations, can’t forget to dispose
- Use-after-disposal?
  - Yes. Local variable has gone out of scope
- Double-disposal?
  - Yes. Local variables don’t go out of scope twice.

Not a Panacea

There are other failure modes that RAII is not intended to solve:

- Resource loops
- Deadlocks

Implementing a RAII class

If the resource that you are trying to wrap is represented as a pointer already, then you do not have to implement your own RAII class. `std::unique_ptr` is likely already able to manage the pointer for you.

Custom Disposal method

Let’s assume that we want to manage a normal FILE.

```
FILE * fopen(const char * filename, const char * mode);
int fclose(FILE * stream);
```

FILE unique_ptr

Now I'm going to present a little boilerplate to make using this a little easier. First, a functor which will be used to dispose of the FILE handle:

```
struct file_closer {
   void operator()(FILE * stream) const { fclose(stream); }
};
```

Then, a using declaration to make my new RAII type:

```
using cfile = std::unique_ptr<FILE, file_closer>;
```

FILE unique_ptr (C++20)

Or: in C++20 you can more simply do:

```
using cfile = std::unique_ptr<FILE,
                    decltype([](FILE * fp){ fclose(fp); })>;
```

Acquiring a file_handle

It would probably be nice if we wrote a factory function to help with opening files:

```
auto make_cfile(char const * filename, char const * mode) {
   FILE * stream{fopen(filename, mode)};
    if (not stream) {
        throw std::runtime_exception{ “Failed to open file” };
    }
    return cfile{stream};
}
```

Using the file_handle

Now we can put this all together:

```
void fn() {
    auto file{make_file("filename.txt", "w")};
    fprintf(file.get(), "Data for the file");
}
```

Shared resource handle

`std::shared_ptr` has a similar custom disposal method mechanism. The same techniques we just talked about applies here, with the added feature that you have a reference-counted resource.

Writing your own RAII class

When writing your own RAII class, there are some design questions that you will need to ask

Is there a valid default acquisition?

- Provide a default constructor to set that up.
- Does not preclude an empty state as well.
- Destructor may need to understand that the resource was released "early"

Is there a valid “empty” state?

- Example: `nullptr` for `std::unique_ptr`
- Perhaps provide a default constructor to set that up instead of (perhaps) a default acquisition
- Ensure that your destructor will do the right thing for an empty resource

Is adopting a resource allowed?

- Provide a single parameter constructor (probably explicit) to set that up.
- Does not preclude an empty state as well.
- Destructor may need to understand that the resource was released "early"

Copyable?

- If not, `= delete` your copy constructor, and your copy assignment operator
- For example, `std::shared_ptr` is copyable, `std::unique_ptr` is not.

Movable?

- If not, `= delete` your move constructor, and your move assignment operator
- Both `std::shared_ptr` and `std::unique_ptr` are movable. `std::scoped_lock` is not.

Access underlying representation?

- Provide a `.get()` or `.native_handle()` method to get the raw representation
- `std::jthread` has one, `std::scoped_lock` does not

Hide the underlying representation?

- Provide public member functions to expose the functionality desired without exposing the representation
- You may still want to allow access to the underlying representation for cases you haven't considered

Dependent Resources

- Provide acquisition functions which returns another RAII class instance to manage that dependent resource

Release the resource?

- Provide a `.release()` method to get the raw representation and release control
- Mark the `.release()` method as `[[nodiscard]]`

Example RAII class

```
template <class Mutex>
class unique_unlock {
  public:
    explicit unique_unlock(std::unique_lock<Mutex> & p_lock)
            : lock(p_lock) { lock.unlock(); }
    // Delete the copy and move constructors and assignment
    // operators
    ~unique_unlock() { lock.lock(); }
  private:
    std::unique_lock<Mutex> & lock;
};
```

Example RAII class usage

```
std::mutex mut;

void fn() {
  std::unique_lock ul{mut};
  // Do some work protected by the mutex
  {
    unique_unlock unl{ul};
    // Do some work not protected by the mutex
  }
  // Do some more work protected by the mutex again
}
```

Core Guidelines on Scope

Since RAII and object lifetime is so intimately intertwined, some Core Guidelines that apply:

R: Resource management

ES.5: Keep scopes small

ES.20: Always initialize an object

ES.21: Don't introduce a variable (or constant) before you need to use it

ES.22: Don't declare a variable until you have a value to initialize it with

Q&A

RAII: Resource Acquisition Is Initialization

Acquire the resource during construction, or take possession of a resource.

Dispose of the resource during destruction.

Andre Kostur
andre@kostur.net
@AndreKostur