Object-Oriented Programming

RAII

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Programming Idiom

A recurring construct to fix a commonly occurring coding problem

- Not a direct feature of the programming language, but a combination of features to solve the problem

- Typically, this is a design problem

- Related to *design patterns* which we will discuss soon

RAII Idiom

Resource Acquisition Is Initialization

- Holding a resource is tied to an object's lifetime
- Constructor: The only place where the resource is allocated
- Destructor: The only place where the resource is deallocated
- Prevents *resource leaks*, *double-free*, *use when invalid*

RAII Interface

- Constructor - Allocate resource
- Destructor - Deallocate resource
- Access - Direct access to a resource
- Boolean - Check if the resource exists (i.e., has been allocated and is still valid)
- Copy, Assignment - Transfer resource control to new RAII object
- Deallocate - Deallocate resource before Destructor (safely)

RAII: std::ofstream

```cpp
// RAII usage
{
    std::ofstream out("output.md");

    out << "# RAII" << '\n';
}
```

```cpp
// non-RAII usage
{
    std::ofstream out;

    out.open("output.md");

    out << "# RAII" << '\n';

    out.close();
}
```

RAII: std::unique_lock

```cpp
// RAII usage
{
    std::unique_lock<std::mutex> lock(mutex);

    if (terminate)
        stop();
    wait = false;
    cond.notify_one();
    cond.wait(lock);
    read = true;
}
```

```cpp
// non-RAII usage
{
    std::unique_lock<std::mutex> lock(mutex);

    if (terminate)
        stop();
    wait = false;
    cond.notify_one();
    cond.wait(lock);
    read = true;

    lock.unlock();
}
```

Smart Pointers

A wrapper type that makes pointers safer

- Automatic initialization with `nullptr`
- Convertible to `bool` for validity checking
- Automatic destructor call for deallocation of the contained resource

C++ Smart Pointers

Smart Pointer	Status
`std::unique_ptr`	Added C++11
`std::shared_ptr`	Added C++11
`std::auto_ptr`	Deprecated in C++11, Removed in C++17

Pointer Use Cases

Use Case	Solution
Optional types	`std::optional`
Lazy Initialization	About to find out
Existing libraries that use pointers	About to find out

Use Case: Lazy Initialization

Delaying the construction of an object until (or if) it is needed

- Primarily for *resource expensive* objects
- Directly, e.g., `sizeof(Data)`, store a large amount of data
- Use a large number of resources

  File handles

  Network connections

  Memory

  Processes

  Threads

  Synchronization primitives (mutexes, semaphores, etc.)

  Kernel handles

Lazy Initialization with Pointers

```
Data* data = nullptr;

// when actually (or if) needed ...
data = new Data();

// ...

// NOTE: Action based on whether exists or not
if (data) {
    // NOTE: Same as data->stuff,
    // must dereference before using
    std::cout << (*data).stuff;
}

// NOTE: Must deallocate
if (data)
    delete data;
```

std::unique_ptr

```cpp
{
    std::unique_ptr<Data> data;

    // when actually (or if) needed ...
    data.reset(new Data());

    // ...

    // NOTE: Action based on whether exists or not
    if (data) {
        // NOTE: Same as data->stuff,
        // must dereference before using
        std::cout << (*data).stuff;
    }

    // NOTE: Automatic deallocation
}
```

Comparison

```cpp
Data* data = nullptr;

// when actually (or if) needed ...
data = new Data();

// ...

// NOTE: Action based on whether exists or not
if (data) {
    // NOTE: Same as data->stuff,
    // must dereference before using
    std::cout << (*data).stuff;
}

// NOTE: Must deallocate
if (data)
    delete data;
```

```cpp
{
    std::unique_ptr<Data> data;

    // when actually (or if) needed ...
    data.reset(new Data());

    // ...

    // NOTE: Action based on whether exists or not
    if (data) {
        // NOTE: Same as data->stuff,
        // must dereference before using
        std::cout << (*data).stuff;
    }

    // NOTE: Automatic deallocation
}
```

Use Case: Existing library with pointers

- Calculating Complexity

- Uses libxml2

Example: xmlReadFile()

```
xmlDoc* doc(xmlReadFile(filename, nullptr, 0));
if (!doc) {
    return -1;
}

xmlXPathContext* xpathCtx(xmlXPathNewContext(doc));
if (!xpathCtx) {
    xmlFreeDoc(doc);
    return -1;
}

// ...

xmlFreeDoc(doc);
xmlXPathFreeContext(xpathCtx);
```

- **Concerns**
- libxml2 functions, e.g., xmlReadFile
- error handling for srcMLXPathCount()
- deallocate with xmlFreeDoc(), etc. with error handling
- Full Example

Custom Deleter

```
std::unique_ptr<xmlDoc, decltype(&xmlFreeDoc)>
    doc(xmlReadFile(filename, nullptr, 0),
        xmlFreeDoc);
if (!doc) {
    return -1;
}

std::unique_ptr<xmlXPathContext,
                decltype(&xmlXPathFreeContext)>
    xpathCtx(xmlXPathNewContext(doc.get()),
             xmlXPathFreeContext);
if (!xpathCtx)
    return -1;

// ,,,
```

- **Concerns**
- libxml2 functions, e.g., xmlReadFile
- error handling for srcMLXPathCount()
- ~~deallocate with xmlFreeDoc(), etc. with error handling~~
- Full Example
- Example of Full Use

Comparison

```
xmlDoc* doc(xmlReadFile(filename, nullptr, 0));
if (!doc) {
    return -1;
}

xmlXPathContext* xpathCtx(xmlXPathNewContext(doc));
if (!xpathCtx) {
    xmlFreeDoc(doc);
    return -1;
}

// ...

xmlFreeDoc(doc);
xmlXPathFreeContext(xpathCtx);
```

```
std::unique_ptr<xmlDoc, decltype(&xmlFreeDoc)>
    doc(xmlReadFile(filename, nullptr, 0),
        xmlFreeDoc);
if (!doc) {
    return -1;
}

std::unique_ptr<xmlXPathContext,
                decltype(&xmlXPathFreeContext)>
    xpathCtx(xmlXPathNewContext(doc.get()),
             xmlXPathFreeContext);
if (!xpathCtx)
    return -1;

// ,,,
```

Specialization of `default_delete<>`

```cpp
namespace std {
    template<>
    struct default_delete<xmlDoc> {
        void operator()(xmlDoc* doc) {
            xmlFreeDoc(doc);
        }
    };

    template<>
    struct default_delete<xmlXPathContext> {
        void operator()(xmlXPathContext* xpathCtx) {
            xmlXPathFreeContext(xpathCtx);
        }
    };
}
```

```cpp
std::unique_ptr<xmlDoc> doc(xmlReadFile(filename, nullptr, 0));
if (!doc) {
    return -1;
}

std::unique_ptr<xmlXPathContext> xpathCtx(xmlXPathNewContext(doc.get()));
if (!xpathCtx)
    return -1;

// ...
```

Alternative Solutions

```
std::unique_ptr<xmlDoc> doc(xmlReadFile(filename, nullptr, 0));
if (!doc) {
    return -1;
}

std::unique_ptr<xmlXPathContext> xpathCtx(xmlXPathNewContext(doc.get()));
if (!xpathCtx)
    return -1;

// ,,,
```

- Create custom C++ wrapper for every libxml2 function used
- Use `goto` and other constructs in the code

Conclusion

- RAII behavior prevents resource errors without any additional burden on the developer

- Ensures proper behavior

- Makes the resource much easier to work with, e.g., don't have to explicitly `close()`

- **All use of resources should be built with RAII behavior**