RAII

Resource Acquisition Is Initialization or RAII, is a C++ programming technique[1][2] which binds the life cycle of a resource (allocated memory, thread of execution, open socket, open file, locked mutex, database connection—anything that exists in limited supply) to the lifetime of an object.

RAII guarantees that the resource is available to any function that may access the object (resource availability is a class invariant). It also guarantees that all resources are released when the lifetime of their controlling objectends, in reverse order of acquisition. Likewise, if resource acquisition fails (the constructor exits with an exception), all resources acquired by every fully-constructed member and base subobject are released in reverse order of initialization. This leverages the core language features (object lifetime, scope exit, order of initialization and stack unwinding) to eliminate resource leaks and guarantee exception safety. Another name for this technique is *Scope-Bound Resource Management* (SBRM), after the basic use case where the lifetime of an RAII object ends due to scope exit.

RAII can be summarized as follows:

- encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
  - the destructor releases the resource and never throws exceptions;
- always use the resource via an instance of a RAII-class that either
  - has automatic storage duration or temporary lifetime itself, or
  - has lifetime that is bounded by the lifetime of an automatic or temporary object

Move semantics make it possible to safely transfer resource ownership between objects, across scopes, and in and out of threads, while maintaining resource safety.

Classes with open()/close(), lock()/unlock(), or init()/copyFrom()/destroy() member functions are typical examples of non-RAII classes:

```
std::mutex m;

void bad()
{
    m.lock();                     // acquire the mutex
    f();                          // if f() throws an exception, the mutex is never released
    if(!everything_ok()) return; // early return, the mutex is never released
    m.unlock();                   // if bad() reaches this statement, the mutex is released
}

void good()
{
    std::lock_guard<std::mutex> lk(m); // RAII class: mutex acquisition is initialization
    f();                               // if f() throws an exception, the mutex is released
    if(!everything_ok()) return;       // early return, the mutex is released
}                                      // if good() returns normally, the mutex is released
```

The standard library

The C++ library classes that manage their own resources follow RAII: `std::string`, `std::vector`, `std::thread`, and many others acquire their resources in constructors (which throw exceptions on errors), release them in their destructors (which never throw), and don't require explicit cleanup.

In addition, the standard library offers several RAII wrappers to manage user-provided resources:

- `std::unique_ptr` and `std::shared_ptr` to manage dynamically-allocated memory or, with a user-provided deleter, any resource represented by a plain pointer;
- `std::lock_guard`, `std::unique_lock`, `std::shared_lock` to manage mutexes.

References

1. ↑ RAII in Stroustrup's C++ FAQ (http://www.stroustrup.com/bs_faq2.html#finally)
2. ↑ C++ Core Guidelines E.6 "Use RAII to prevent leaks" (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#e6-use-raii-to-prevent-leaks)