WIKIPEDIA

Resource acquisition is initialization

Resource acquisition is initialization (**RAII**)[1] is a programming idiom[2] used in several object-oriented languages. In RAII, holding a resource is a class invariant, and is tied to object lifetime: resource allocation (or acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor. Thus the resource is guaranteed to be held between when initialization finishes and finalization starts (holding the resources is a class invariant), and to be held only when the object is alive. Thus if there are no object leaks, there are no resource leaks.

RAII is associated most prominently with C++ where it originated, but also D, Ada, Vala, and Rust. The technique was developed for exception-safe resource management in C++[3] during 1984–89, primarily by Bjarne Stroustrup and Andrew Koenig,[4] and the term itself was coined by Stroustrup.[5] RAII is generally pronounced as an initialism, sometimes pronounced as "R, A, double I".[6]

Other names for this idiom include *Constructor Acquires, Destructor Releases* (CADRe) [7] and one particular style of use is called *Scope-based Resource Management* (SBRM).[8] This latter term is for the special case of automatic variables. RAII ties resources to object *lifetime,* which may not coincide with entry and exit of a scope. (Notably variables allocated on the free store have lifetimes unrelated to any given scope.) However, using RAII for automatic variables (SBRM) is the most common use case.

Contents

C++11 example

The following C++11 example demonstrates usage of RAII for file access and mutex locking:

```
#include <mutex>
#include <iostream>
#include <string>
#include <fstream>
#include <stdexcept>

void write_to_file (const std::string & message) {
    // mutex to protect file access (shared across threads)
    static std::mutex mutex;

    // lock mutex before accessing file
    std::lock_guard<std::mutex> lock(mutex);

    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;

    // file will be closed 1st when leaving scope (regardless of exception)
    // mutex will be unlocked 2nd (from lock destructor) when leaving
    // scope (regardless of exception)
}
```

This code is exception-safe because C++ guarantees that all stack objects are destroyed at the end of the enclosing scope, known as stack unwinding. The destructors of both the *lock* and *file* objects are therefore guaranteed to be called when returning from the function, whether an exception has been thrown or not.[9]

Local variables allow easy management of multiple resources within a single function: they are destroyed in the reverse order of their construction, and an object is destroyed only if fully constructed—that is, if no exception propagates from its constructor.[10]

Using RAII greatly simplifies resource management, reduces overall code size and helps ensure program correctness. RAII is therefore highly recommended in C++, and most of the C++ standard library follows the idiom.[11]

Benefits

The advantages of RAII as a resource management technique are that it provides encapsulation, exception safety (for stack resources), and locality (it allows acquisition and release logic to be written next to each other).

Encapsulation is provided because resource management logic is defined once in the class, not at each call site. Exception safety is provided for stack resources (resources that are released in the same scope as they are acquired) by tying the resource to the lifetime of a stack variable (a local variable declared in a given scope): if an exception is thrown, and proper exception handling is in place, the only code that will be executed when exiting the current scope are the destructors of objects declared in that scope. Finally, locality of definition is provided by writing the constructor and destructor definitions next to each other in the class definition.

Resource management therefore needs to be tied to the lifespan of suitable objects in order to gain automatic allocation and reclamation. Resources are acquired during initialization, when there is no chance of them being used before they are available, and released with the destruction of the same objects, which is guaranteed to take place even in case of errors.

Comparing RAII with the `finally` construct used in Java, Stroustrup wrote that "In realistic systems, there are far more resource acquisitions than kinds of resources, so the "resource acquisition is initialization" technique leads to less code than use of a "finally" construct."[1]

Typical uses

The RAII design is often used for controlling mutex locks in multi-threaded applications. In that use, the object releases the lock when destroyed. Without RAII in this scenario the potential for deadlock would be high and the logic to lock the mutex would be far from the logic to unlock it. With RAII, the code that locks the mutex essentially includes the logic that the lock will be released when execution leaves the scope of the RAII object.

Another typical example is interacting with files: We could have an object that represents a file that is open for writing, wherein the file is opened in the constructor and closed when execution leaves the object's scope. In both cases, RAII ensures only that the resource in question is released appropriately; care must still be taken to maintain exception safety. If the code modifying the data structure or file is not exception-safe, the mutex could be unlocked or the file closed with the data structure or file corrupted.

Ownership of dynamically allocated objects (memory allocated with `new` in C++) can also be controlled with RAII, such that the object is released when the RAII (stack-based) object is destroyed. For this purpose, the C++11 standard library defines the smart pointer classes `std::unique_ptr` for single-owned objects and `std::shared_ptr` for objects with shared ownership.

Similar classes are also available through `std::auto_ptr` in C++98, and `boost::shared_ptr` in the Boost libraries.

Clang and GCC "cleanup" extension for C

Both Clang and GNU Compiler Collection implement a non-standard extension to the C language to support RAII: the "cleanup" variable attribute.[12] The following macro annotates a variable with a given destructor function that it will call when the variable goes out of scope:

```
static inline void fclosep(FILE **fp) { if (*fp) fclose(*fp); }
#define _cleanup_fclose_ __attribute__((cleanup(fclosep)))
```

This macro can then be used as follows:

```
void example_usage() {
  _cleanup_fclose_ FILE *logfile = fopen("logfile.txt", "w+");
  fputs("hello logfile!", logfile);
}
```

In this example, the compiler arranges for the *fclosep* function to be called before *example_usage* returns.

Limitations

RAII only works for resources acquired and released (directly or indirectly) by stack-allocated objects, where there *is* a well-defined static object lifetime. Heap-allocated objects which themselves acquire and release resources are common in many languages, including C++. RAII depends on heap-based objects to be implicitly or explicitly deleted along all possible execution paths, in order to trigger its resource-releasing destructor (or equivalent).[13]:8:27 This can be achieved by using smart pointers to manage all heap objects, with weak-pointers for cyclically referenced objects.

In C++, stack unwinding is only guaranteed to occur if the exception is caught somewhere. This is because "If no matching handler is found in a program, the function terminate() is called; whether or not the stack is unwound before this call to terminate() is implementation-defined (15.5.1)." (C++03 standard, §15.3/9).[14] This behavior is usually acceptable, since the operating system releases remaining resources like memory, files, sockets, etc. at program termination.

Reference counting

Perl, Python (in the CPython implementation),[15] and PHP[16] manage object lifetime by reference counting, which makes it possible to use RAII. Objects that are no longer referenced are immediately destroyed or finalized and released, so a destructor or finalizer can release the resource at that time. However, it is not always idiomatic in such languages, and is specifically discouraged in Python (in favor of context managers and *finalizers* from the *weakref* package).

However, object lifetimes are not necessarily bound to any scope, and objects may be destroyed non-deterministically or not at all. This makes it possible to accidentally leak resources that should have been released at the end of some scope. Objects stored in a static variable (notably a global variable) may not be finalized when the program terminates, so their resources are not released; CPython makes no guarantee of finalizing such objects, for instance. Further, objects with circular references will not be collected by a simple reference counter, and will live indeterminately long; even if collected (by more sophisticated garbage collection), destruction time and destruction order will be non-deterministic. In CPython there is a cycle detector which detects cycles and finalizes the objects in the cycle, though prior to CPython 3.4, cycles are not collected if any object in the cycle has a finalizer.[17]

References

1. Bjarne Stroustrup Why doesn't C++ provide a "finally" construct? (http://www.stroustrup.com/bs_faq2.html#finally) Accessed on 2013-01-02.

2. Sutter, Herb; Alexandrescu, Andrei (2005). *C++ Coding Standards*. C++ In-Depth Series. Addison-Wesley. p. 24. ISBN 0-321-11358-6.
3. Stroustrup 1994, 16.5 Resource Management, pp. 388–89.
4. Stroustrup 1994, 16.1 Exception Handling: Introduction, pp. 383–84.
5. Stroustrup 1994, p. 389. I called this technique "resource acquisition is initialization."
6. "How do you pronounce RAII? (https://stackoverflow.com/questions/99979/how-do-you-pronounce-raii)", *StackOverflow*
7. https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/UnarLCzNPcI
8. http://allenchou.net/2014/10/scope-based-resource-management-raii/
9. "dtors-shouldnt-throw" (http://www.parashift.com/c++-faq/dtors-shouldnt-throw.html). Retrieved 12 February 2013.
10. "Working Draft, Standard for Programming Language C++" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf) (PDF). Retrieved 28 December 2017.
11. "too-many-trycatch-blocks" (https://web.archive.org/web/20141101155411/http://www.parashift.com/c++-faq/too-many-trycatch-blocks.html). *C++ FAQ*. Archived from the original (http://www.parashift.com/c++-faq/too-many-trycatch-blocks.html) on 1 November 2014. Retrieved 4 June 2014.
12. https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html
13. Weimer, W; Necula, G.C. (2008). "Exceptional Situations and Program Reliability" (http://www.cs.virginia.edu/~weimer/p/weimer-toplas2008.pdf) (PDF). *ACM Transactions on Programming Languages and Systems, vol 30 (2)*.
14. "RAII and Stack unwinding" (https://stackoverflow.com/questions/5557555/raii-and-stack-unwinding). 2011.
15. Extending Python with C or C++ (https://docs.python.org/extending/extending.html): Reference Counts (https://docs.python.org/extending/extending.html#reference-counts)
16. https://stackoverflow.com/a/4938780
17. `gc` — Garbage Collector interface (https://docs.python.org/3/library/gc.html)

Further reading

- Stroustrup, Bjarne (1994). *The Design and Evolution of C++*. Addison-Wesley. ISBN 0-201-54330-3.

External links

- Sample Chapter "Gotcha #67: Failure to Employ Resource Acquisition Is Initialization (http://www.informit.com/articles/article.aspx?p=30642&seqNum=8)" by Stephen Dewhurst
- Interview "A Conversation with Bjarne Stroustrup (http://artima.com/intv/modern3.html)" by Bill Venners
- Article "The Law of The Big Two (http://artima.com/cppsource/bigtwo3.html)" by Bjorn Karlsson and Matthew Wilson
- Article "Implementing the 'Resource Acquisition is Initialization' Idiom (https://web.archive.org/web/20080129235233/http://gethelp.devx.com/techtips/cpp_pro/10min/2001/november/10min1101.asp)" by Danny Kalev
- Article "RAII, Dynamic Objects, and Factories in C++ (http://www.codeproject.com/KB/cpp/RAIIFactory.aspx)" by Roland Pibinger
- RAII in Delphi "One-liner RAII in Delphi (http://blog.barrkel.com/2010/01/one-liner-raii-in-delphi.html)" by Barry Kelly