RAII in C++

10/10/2019 By An7ar35

Contents

1 Definition

Resource **A**quisition **I**s **I**nitialization essentially describes the concept that a resource held by an object is tied to the lifetime of said object. It is an incredibly useful and powerful concept that allows automatic cleaning of resources without the need for Garbage Collection (like in Java for example).

During an object's construction the resources[1] are allocated (or acquired) by the constructor and are held by the object until its destruction stage where they are released in reverse order (see appendix A for an example).

RAII, with code discipline, helps avoid resource leaks and guaranties exception safety as long as these resources are acquired & released from stack-allocated objects where their lifetime is well defined. The stack's guarantee to unwind, and thus trigger release for the resources, is only held if exceptions are caught[2].

1.1 Scope-based Resource Management (SBRM)

Scope-based Resource Management leverages RAII in the sense that when we exit the scope where a resource is managed, the resource is released. So, for example, in a function where a smart pointer is created and used without being returned its destructor will be called when the function is existed. Scope can also be artificially forced by using brackets `{...}` (see appendix B for an example).

1.2 Heap allocated objects

Any resources on the heap (a.k.a. the free-store) created can live beyond the lifetime of its creator object so some care is required when dealing with them. Implicit/Explicit deletion must be implemented and always called when the resource is not required any longer.

1.2.1 Automatic management (smart pointers)

A `std::unique_ptr` manages the lifetime of the pointed object based on the lifetime of its owner. Even though pointer ownership can be explicitly transferred, the resources is tied to its pointer (see appendix C for an example).

Alternatively, in cases where a resource needs to be shared, a `std::shared_ptr`) can be used instead. Release is done by whatever is the last remaining `std::shared_ptr` pointing to the resource (see appendix D for an example). In the case of cyclically referenced objects, weak pointer can be used along side a shared pointer.

[1] Raw pointers pose some issues here (see section 1.2).

[2] Uncaught exceptions will trigger the `terminate()` function early leaving any resources acquired/allocated in a unreleased state.

1.2.2 Manual management (raw pointers)

Don't if it can be avoided. Use smart pointers. If there's no way around using raw pointers then some manual intervention in order to make sure any resources pointed to are properly released and avoid leaks will be required. Also ownership should remain clear. Clear and simple code leads to less headaches down the road.

Keeping the scope of the pointer usage within a container object's lifetime is advised so that the object's destructor can be used to free up the resources pointed to (see appendix E for an example).

2 Some common usage scenarios

2.1 Mutex locks

In multi-threaded development, RAII can be used to control mutex locks where a locked object is unlocked when the lock itself is destroyed. This makes sense as without a lock something cannot be locked any-longer! :)

The example taken from cppreference.com shows how to approach RAII mutex locks:

Non-RAII & unsafe mutex locking

```
1 std::mutex m;
2 void bad() {
3     m.lock();
4     f();
5     if( !everything_ok() )
6         return;
7     m.unlock();
8 }
```

Here the mutex is acquired (line 4) and released (line 8) but may create a deadlock in the following scenarios:

1. `f()` throws an exception (line 5),
2. `everthing_ok()` evaluates to `false` and triggers an early return from the function (line 6-7).

RAII & safe mutex locking

```
1 void good() {
2     std::lock_guard<std::mutex> lk( m );
3     f(); released
4     if( !everything_ok() )
5         return;
6 }
```

By using an RAII class (`std::lock_guard<>`) that deals with the locking / unlocking at construction / destruction it means that the deadlocks present in the previous unsafe example are dealt with.

2.2 Files

RAII can be especially useful when accessing files whether to read from or write to them. A file can be open on construction of a 'handling' object and closed when this object goes out of scope and its destructor is called. The only thing left is then to deal with any potential exceptions that might be raised lest corruption occur to the data being written or read.

RAII file access

```
void write( const std::string& line, const std::string &file_name ) {
    static std::mutex mutex;
    std::lock_guard<std::mutex> lock( mutex ); //lock mutex for file access
    std::ofstream file( file_name );

    if( file.is_open() )
        throw std::runtime_error( "Could not open file: " + file_name );

    file << line << std::endl;
}
```

Appendices

The appendices collate minimal code examples of RAII in action.

A Example: Nested class construction/destruction

Nested class example

```cpp
#include <iostream>

class A {
  public:
    A()  { std::cout << "Class A constructor called." << std::endl; };
    ~A() { std::cout << "Class A destructor called."  << std::endl; };
};

class B {
  public:
    B()  { std::cout << "Class B constructor called." << std::endl; };
    ~B() { std::cout << "Class B destructor called."  << std::endl; };
};

class Container {
  public:
    Container()  { std::cout << "Container constructor called." << std::endl; };
    ~Container() { std::cout << "Container destructor called."  << std::endl; };

  private:
    A a;
    B b;
};

int main() {
    auto c = Container();
}
```

Output:

```
Class A constructor called.
Class B constructor called.
Container constructor called.
Container destructor called.
Class B destructor called.
Class A destructor called.
```

B Example: Scope-based Resource Management

Scope based memory management example

```cpp
#include <iostream>
#include <memory>

class A {
  public:
    A( int id ) : _id( id ) { std::cout << "Class A #" << _id << " constructor called."    st
    ~A() { std::cout << "Class A #" << _id << " destructor called."  << std::endl; };

  private:
    int _id;
};

void foo() { //function scope begins
    std::cout << "Entering Foo()" << std::endl;
    auto a1 = A( 1 );
    { //force-scopping
        auto a2 = A( 2 );
    }
    { //force-scopping
        auto a3 = A( 3 );
    }
    std::cout << "Exiting Foo()" << std::endl;
} //function scope ends

int main() {
    foo();
}
```

Output:

```
Entering Foo()
Class A #1 constructor called.
Class A #2 constructor called.
Class A #2 destructor called.
Class A #3 constructor called.
Class A #3 destructor called.
Exiting Foo()
Class A #1 destructor called.
```

C Example: Managed heap-allocated objects (unique_ptr)

Heap objects with shared pointers example 1

```cpp
#include <iostream>
#include <memory>

class A {
  public:
    A()  { std::cout << "Class A constructor called." << std::endl; };
    ~A() { std::cout << "Class A destructor called."  << std::endl; };

    int add( int a, int b ) {
        return a + b;
    }
};

void foo() {
    std::cout << "Entering Foo()" << std::endl;
    auto p = std::make_unique<A>();
    std::cout << "3 + 4 = " << p->add( 3, 4 ) << std::endl;
    std::cout << "Exiting Foo()" << std::endl;
}

int main() {
    foo();
}
```

Output:

```
Entering Foo()
Class A constructor called.
3 + 4 = 7
Exiting Foo()
Class A destructor called.
```

D Example: Managed heap-allocated objects (shared_ptr)

Heap objects with shared pointers example 2

```cpp
#include <iostream>
#include <memory>

class A {
  public:
    A( int id ) : _id( id ) {
        std::cout << "Class A #" << _id << " constructor called." << std::endl;
    };

    ~A() {
        std::cout << "Class A #" << _id << " destructor called."  << std::endl;
    };

    int getID() const { return _id; };

  private:
    int _id;
};

void bar( std::shared_ptr<A> ptr ) {
    std::cout << "Entering Bar()" << std::endl;
    std::cout << "A::getID(): " << ptr->getID() << std::endl;
    std::cout << "Exiting Bar()" << std::endl;
}

void foo( std::shared_ptr<A> ptr ) {
    std::cout << "Entering Foo()" << std::endl;
    auto b = std::make_shared<A>( 2 );
    bar( ptr );
    bar( b );
    std::cout << "Exiting Foo()" << std::endl;
}

int main() {
    std::cout << "Entering main()" << std::endl;
    auto a = std::make_shared<A>( 1 );
    foo( a );
    std::cout << "Exiting main()" << std::endl;
}
```

Output:

```
Entering main()
Class A #1 constructor called.
```

```
Entering Foo()
Class A #2 constructor called.
Entering Bar()
A::getID(): 1
Exiting Bar()
Entering Bar()
A::getID(): 2
Exiting Bar()
Exiting Foo()
Class A #2 destructor called.
Exiting main()
Class A #1 destructor called.
```

E Example: RAII with raw pointers

Heap objects with raw pointers example

```
1  #include <iostream>
2
3  struct Info {
4      int _id;
5      std::string _name;
6  };
7
8  class A {
9    public:
10     A( int id, std::string name ) : _info_ptr( new Info( { id, name } ) ) {
11         std::cout << "Class A constructor called." << std::endl;
12     };
13
14     ~A() {
15         std::cout << "Class A destructor called."  << std::endl;
16         delete _info_ptr; //free resource
17     };
18
19     void printInfo() const {
20         std::cout << "id: " << _info_ptr->_id << ", name: " << _info_ptr->_name << std::endl
21     }
22
23   private:
24     Info * _info_ptr;
25 };
26
27 int main() {
28     int id_count = 0;
29     auto a = A( 25, "John" );
30     a.printInfo();
31 }
```

Output:

```
Class A constructor called.
id: 25, name: John
Class A destructor called.
```

Line 16 releases the memory resource taken by the Info object the pointer is pointing to. If it was omitted then the memory would still be marked as used but when the A object was destroyed there would be no way to access and thus free that resource any longer. Therefore there would be a memory leak.