- Example

The example in this lesson shows the deterministic behavior of RAII in C++.

WE'LL COVER THE FOLLOWING

Example - RAII

RAII stands for **R**esource **A**cquisition **I**s **I**nitialization, and it is one of the most important idioms in C++. It states that a resource should be acquired in the constructor of the object and released in the destructor of the object. It is important to remember that the destructor will automatically be called if the object goes out of scope.

Is this not deterministic? In Java or Python (`__del__`), you have a destructor but not the guarantee. Therefore, if you use the destructor to release a critical resource, such as a lock, it can end disastrously. In C++, however, this problem is prevented.

Look at the example below:

```
// raii.cpp

#include <iostream>
#include <new>
#include <string>

class ResourceGuard{
  private:
    const std::string resource;
  public:
    ResourceGuard(const std::string& res):resource(res){
      std::cout << "Acquire the " << resource << "." <<  std::endl;
    }
    ~ResourceGuard(){
      std::cout << "Release the "<< resource << "." << std::endl;
    }
};


int main(){

  std::cout << std::endl;

  ResourceGuard resGuard1{"memoryBlock1"};

  std::cout << "\nBefore local scope" << std::endl;
  {
    ResourceGuard resGuard2{"memoryBlock2"};
  }
  std::cout << "After local scope" << std::endl;

  std::cout << std::endl;


  std::cout << "\nBefore try-catch block" << std::endl;
  try{
      ResourceGuard resGuard3{"memoryBlock3"};
      throw std::bad_alloc();
  }
  catch (std::bad_alloc& e){
      std::cout << e.what();
  }
  std::cout << "\nAfter try-catch block" << std::endl;

  std::cout << std::endl;

}
```

Explanation

- `ResourceGuard` is a guard that manages its resource. In this case, the resource is a simple string. `ResourceGuard` reliably creates its resource and releases the resource in its destructor (line 14 - 16).

- The destructor of `resGuard1` (line 23) will be called exactly at the end of the `main` function (line 46).

- The lifetime of `resGuard2` (line 27) ends in line 28. Therefore, the destructor will automatically be executed. Even the presence of an exception does not alter the reliability of `resGuard3` (line 36).

- Its destructor will be called at the end of the `try` block (line 35 - 38).

Let's test your understanding of this example with an exercise in the next lesson.