- Example

In this lesson, we'll see a few examples of memory allocation and deallocation in C++.

WE'LL COVER THE FOLLOWING

RAII

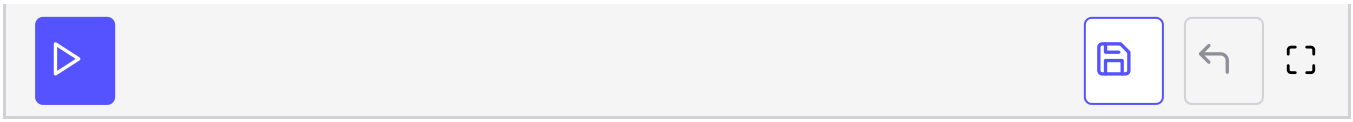
RAII stands for **R**esource **A**cquisition **I**s **I**nitialization. Probably the most important idiom in C++ says that a resource should be acquired in the constructor of the object and released in the destructor of the object. The key is that the destructor will automatically be called if the object goes out of scope.

Is it not totally deterministic? In Java or Python (`__del__`), we have a destructor but not the guarantee. Therefore, if we use the destructor to release a critical resource like a lock, it can end disastrously. But in C++, we are safe.

See the example below:

```
#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 resources. In this case, the resource is a simple string. `ResourceGuard` creates, in its constructor (lines 9 - 11), the resource and releases it in its destructor (lines 12 - 14). It does its job very reliably.
- The destructor of `resGuard1` (line 21) will be called exactly at the end of the `main` function (line 44).
- The lifetime of `resGuard2` (line 25) already ends in line 26. Therefore, the destructor will automatically be executed as soon as it goes out of scope on line 26.
- Even the throwing of an exception does not alter the reliability of `resGuard3` (line 34). Its destructor will be called at the end of the `try` block (lines 33 - 36).

Next, we'll learn how to overload `new` and `delete`.