Destructors

Let's understand the functionality of destructors.

WE'LL COVER THE FOLLOWING

As the name suggests, a **destructor** is the opposite of a constructor.

> The purpose of a destructor is to destroy a class object after it goes out of scope. This frees up the memory previously occupied by the object.

Properties

- Unlike the constructor, a destructor is called automatically.
- We can define a destructor inside or outside the class. In its body, special actions, such as freeing up memory or releasing locks, can be performed.
- A destructor can be declared by the `~` operator followed by the name of the class: `~Account()`
- It does not have a return type or any parameters.
- A class's destructor cannot be overloaded.
- Apart from being automatically triggered, an explicit call to `delete` or `delete []` triggers the destructor.

Rules

We can define a destructor if a class needs an explicit action at object destruction. To be more precise, the destructor of the object is invoked when the object goes out of scope. Because of this totally deterministic behavior, we can release highly critical resources in the destructor.

Locks or smart pointers in C++ use this characteristic. Both will automatically release their underlying resources if they go out of scope.

```
void func(){
  std::unique_ptr<int> uniqPtr = std::make_unique<int>(2011);
  std::lock_guard<std::mutex> lock(mutex);
  ...
} // automatically released
```

We can also read this rule the other way around. If all the members of our class have a default destructor, we should not define our own.

All resources acquired by a class must be released by the class's destructor. This rule sounds quite obvious and helps us to prevent resource leaks. Right? But we have to consider which of our class members have a full set of default operations. Now we are once more back to the rule of zero or five.

Maybe the `File` class has no destructor in contrast to `std::ifstream` and, therefore, we may get a memory leak if instances of `MyClass` go out of scope.

```
class MyClass{
    std::ifstream fstream;   // may own a file
    File* file_;             // may own a file
    ...
};
```

If a class has a raw pointer (`T*`) or reference (`T&`), consider whether it might be owned. This is a question we have to answer if our class has raw pointers or references.

Who is the owner? If our class is the owner, we have to delete the resource. In other words, if a class owns a pointer or reference member, define a destructor. In such a case, it's a good choice to use a smart pointer like `std::unique_ptr`.

`std::unique_ptr` is by design as efficient as a raw pointer can be. So we have

`std::unique_ptr` is by design as efficient as a raw pointer can be. So we have no overhead in time or memory, only added value.

A base class destructor should be either public and virtual or protected and nonvirtual. This is quite easy to get. If the destructor of the base class is protected, we cannot destroy derived objects using a base class pointer; therefore, the destructor must not be virtual.

To specifically clarify types (not pointers or references):

If the destructor of a class, `Base`, is private, we cannot use the type. If the destructor of a class, `Base`, is protected, we can only derive `Derived` from Base and use Derived.

```
struct Base{
    protected:
    ~Base() = default;
};

struct Derived: Base{};

int main(){
    Base b;   // Error: Base::~Base is protected within this context
    Derived d;
}
```

Calling `Base b` will cause an error.

Further information

- The rule of zero or five

Let's look at an example of destructors in the next lesson.