ToiletPaper #96

C++ Destructors Are the Best

Author: Alexandros Panagiotidis / Senior Software Architect / Office Stuttgart

Destructors are methods in object-oriented programming languages that – as their name suggests – are automatically called at the end of an objects' lifecycle. Traditionally, these are used to release resources that were allocated during the objects' lifetime. For objects allocated on the stack, destructors are called in the reverse order of their initialization when leaving the scope. This guarantee of the C ++ standard can be used for some interesting things:

```
struct SafeTransaction {
  // Keep non-owning reference to database
  SafeTransaction(Database & db) : db_{ db },
commited_{ false } {
    db_.begin_transaction();
  }
  ~SafeTransaction() {
    if (!commited_) { // Commit if not already done
      db_.commit_transaction();
    }
  }
  void commit() { // Allow early commit
    db_.commit_transaction();
    commited_ = true;
  }
  // ...
}
```

```
struct StateGuard {
  // Keep copy of previous state
  StateGuard() : oldState_{ fetchState() } {
  }
  // Restore previous state
  ~StateGuard() {
    setState(oldState_);
  }
  // ...
}
```

This pattern is called **Resource Acquisition is Initialization** (RAII) and ensures that instances of resources can exist only if they have been properly initialized and are properly cleaned up again. When implementing such classes, you should think about whether you also need the **Rule-of-Three/-Five** and if you handle exceptions in the destructor.

➔ Examples RAII in Java

Although there are no destructors in Java, it is possible to simulate RAII-like behavior. Theoretically, you could use Object.finalize, which is explicitly intended to clean up resources. However, it is called by the garbage collector when an object is removed from memory. It is therefore not clear, when *finalize* is called and therefore not equivalent to how destructors work with regard to RAII. As an alternative, you can use the *AutoCloseable* interface:

```
class Context implements AutoCloseable {
  private String prefix;
  Context(String prefix) {
    this.prefix = prefix;
    logger.info("Entering {}", this.prefix);
  }
  @Override
  void close() {
    logger.info("Leaving {}", this.prefix);
  }
  // ...
}
```

```
public class Test {
  public static void main() {
    try (Context ctx = new Context("main()")) {
      ctx.log("Hello jambit!");
    }
  }
  public static void something() {
    try (Context ctx = new Context("something()")) {
      ctx.log("Where innovation works");
    }
  }
}
```

When exiting the try block, the try-with-resources block ensures that *close* is called. As a result, you get "real" RAII behavior. Although try-with-resources is already available since Java 7, you need at least Java 8, because *AutoCloseable* has not been introduced before that.

➔ Finally in C++

Going "full-circle", it is worth mentioning that you can build something similar to *finally* since C ++ 11. The specific implementation is left to the inclined reader as practice, but you can cheat by checking the Guideline Support Library. However, you should pay heed to the advice of Herb Sutter (appropriate place in the talk at about 22 minutes) that *finally* is usually not necessary in C ++ if you implement your classes correctly according to the RAII principle.

+ Further Aspects

- More about Resource Acquisition Is Initialization: Wikipedia, cppreference
- More about the Rule-of-Three/-Five: Wikipedia, cppreference
- Youtube: How to Adopt Modern C++17 into Your C++ Code (Herb Sutter, Build 2018)
- Guideline Support Library (GSL), gsl::final_action
- Java: Object.finalize, try-with-resources, AutoCloseable