# ToiletPaper #96

## C++ Deschdrukdora sinds Beschde

Autor: Alexandros Panagiotidis / Senior Software Architect / Standort Stuttgart

Destruktoren sind Methoden in objektorientierten Programmiersprachen die - wie ihr Name vermuten lässt - am Ende des Objektlebenszyklus automatisch aufgerufen werden. Traditionell nutzt man diese, um Ressourcen freizugeben, die man während der Lebenszeit des Objekts allokiert hat. Bei stack-allokierten Objekten werden Destruktoren beim Verlassen des Scopes in umgekehrter Reihenfolge ihrer Initialisierung aufgerufen. Diese Garantie des C++-Standards kann man für einige interessante Dinge nutzen:

```
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_);
  }
  // ...
}
```

Dieses Pattern nennt sich **Resource Acquisition is Initialization** (RAII) und sorgt im Wesentlichen dafür, dass Instanzen von Ressourcen nur existieren können, wenn diese korrekt initialisiert wurden und dann auch wieder korrekt aufgeräumt werden. Bei der Implementation solcher Klassen sollte man in sich gehen und nachdenken, ob man nicht auch die **Rule-of-Three /Five** benötigt und ob man Exceptions im Destruktor behandelt.

## ➔ Beispiel RAII in Java

Obwohl es in Java keine Desktruktoren gibt, kann man RAII-ähnliches Verhalten simulieren. Theoretisch könnte man dafür Object.finalize verwenden, welches explizit zum Aufräumen von Ressourcen vorgesehen ist. Dieses wird jedoch vom Garbage Collector aufgerufen, wenn es ein Objekt aus dem Speicher räumt. Es ist daher nicht klar, wann *finalize* aufgerufen wird und daher nicht equivalent zur Funktionsweise von Destruktoren bzgl. RAII. Eine Alternative bietet die Nutzung des *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");
    }
  }
}
```

Der try-resource-Block stellt sicher, dass *close* beim Verlassen des try-Blocks aufgerufen wird und man daher "echtes" RAII-Verhalten erhält. Obwohl try-resource bereits seit Java 7 verfügbar ist, benötigt man mindestens Java 8, da erst dann *AutoCloseable* eingeführt wurde.

## ➔ Finally in C++

Um den "Kreis" wieder zu schließen sei an dieser Stelle noch kurz erwähnt, dass man sich seit C++11 ein *finally*-Konstrukt basteln kann. Die konkrete Implementierung bleibt dem geneigten Leser zur Übung überlassen, aber kann man in der Guideline Support Library spickeln. Dabei sollte man jedoch den Rat von Herb Sutter beherzigen (passende Stelle im Talk bei ca. 22 Minuten), dass *finally* in C++ meist nicht notwendig ist, wenn man seine Klassen gleich nach dem RAII-Prinzip implementiert.

## ✚ Weiterführende Aspekte

- Mehr über Resource Acquisition Is Initialization: Wikipedia,cppreference
- Mehr über 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