# Beispiel `06_sync1`

Dr. Günter Kolousek

15. Oktober 2018

## 1 Allgemeines

- **Backup** nicht vergessen!
- Im ersten Beispiel gibt es genaue Anweisungen zum Aufbau und der Durchführung eines Beispiels. Bei Bedarf nochmals durchlesen!
- Trotzedem hier noch zwei Erinnerungen:
  - Regelmäßig Commits erzeugen!
  - Backup deines Repos nicht vergessen (am Besten nach jedem Beispiel)!!!
- Verwende das bereitgestellte Archiv `template.tar.gz` zum Erstellen eines Meson-Projektes. Es enthält die notwendigen Anpassungen zur Verwendung von Threads.
- Die Anpassungen der Datei `.hgignore` sollten schon erledigt sein, sodass das Verzeichnis `build` nicht versioniert wird.
- In diesem Sinne ist jetzt ein neues Verzeichnis `06_sync1` anzulegen.

## 2 Aufgabenstellung

Dieses Beispiel hat den Sinn auf die verschiedenen Synchronisationsprobleme hinzuweisen.
Los geht's!

## 3 Anleitung

1. Schreibe eine Klasse `Account`, die über einen Kontostand `balance`, eine entsprechende Getter-Methode `int get_balance()` sowie über die Methoden `void deposit(int amount)` und `bool withdraw(int amount)` verfügt. Diese Methoden sollen lediglich von dem Konto (unsynchronisiert) etwas abheben (also so etwas

wie `balance -= amount;` oder einzahlen. Abheben sollte nur erlaubt sein, wenn der Kontostand positiv bleibt. Konnte das Abheben durchgeführt werden, dann liefert `withdraw()` `true` zurück, anderenfalls `false`.

Erstelle ein Modul `account`, d.h. es sind die Dateien `account.h` und `account.cpp` zu erstellen. Die `.h` Dateien kommen in das Verzeichnis `include` und die `.cpp` Dateien in das Verzeichnis `src`. Zu Übungszwecken soll für die Klasse `Account` alle Methoden (member functions) in `account.cpp` erstellt werden.

Erstelle weiters eine Datei `main.cpp`, die eine Instanz der Klasse `Account` anlegt und die Funktionalität dieser Klasse im single-threaded Betrieb zeigt (also ein paar Testausgaben reichen für diese einfache Klasse).

**Übungszweck**

- Modul erstellen

2. Schreibe jetzt das Programm so um, dass das Konto mit 1€ initialisiert wird und 2 Threads gestartet werden, die jeweils 1€ abheben wollen. Diese Threads sollen mittels Lambdaausdrücken realisiert werden.

   Der alte Code soll auskommentiert werden und mit so etwas wie "Punkt 1" markiert werden.

   Wie sieht das Ergebnis aus? Alles wie erwartet?

   **Übungszweck**

   - Threads mit Lambdaausdruck realisieren.

3. Baue die Klasse `Account` jetzt um, sodass zwischen der Abfrage und dem eigentlichen Abbuchen dem jeweils anderen Thread eine Chance gegeben wird, weiterzumachen. Das kann mittels `this_thread::yield()` erreicht werden.

   Wie sieht das Ergebnis jetzt aus? Alles wie erwartet?

   **Übungszweck**

   - `this_thread::yield()` kennenlernen
   - Race conditions verstehen

4. Schreibe jetzt eine eigene Klasse `Depositer`, die eine Instanz von Account (als Referenz) bekommt und als Thread gestartet in einer Schleife jeweils 5 Mal einen Euro aufbucht. Diese Klasse kann zum Modul `account` hinzugefügt werden.

   Bei der Implementierung des Konstruktors beachte, dass die Initialisierung einer Instanzvariable durch einen Parameter am Besten in der "initializer list" vorgenommen wird. Die Implementierung dieser Klasse soll jetzt in der Headerdatei vorgenommen werden. Was ist der Unterschied zur Implementierung in einer Headerdatei vs. der Implementierung in einer `.cpp` Datei?

Das Konto wird jetzt mit 0 initialisiert. Starte zwei Threads mit je einer Instanz von `Depositer` und als Ergebnis des Programmes wirst du vermutlich 10 zu Gesicht bekommen. Auch hier soll der alte Programmcode auskommentiert und entsprechend markiert werden. Ist das erwartet? Ja, wirklich?

Möglich, aber baue jetzt im Schleifenrumpf direkt vor dem Aufbuchen eine kleine Wartezeit, wie z.B. `sleep_for(100ms * i);` ein und starte das Programm wiederum! Und gleich wieder und wieder! Hmm, warum das?

Lege jetzt eine kleine Denkpause ein. Ja, woran liegt das? Ok, es liegt daran, dass ein `balance += amount` **keine** atomare Operation ist, sondern vom Prinzip nichts anderes als `balance = balance + amount;` ist. Durch das `sleep_for` haben wir nur dem Scheduler ein bisserl unter die Arme gegriffen und damit wiederum diesen Fehler provoziert! Ok, `yield()` hätte es auch getan, aber immer die gleichen Mitteln einzusetzen hat ja keinen Lerneffekt, obwohl das `yield()` schon besser ist, nicht wahr?

**Übungszweck**

- Initializerliste üben
- Implementierung in Headerdatei vs. in Quelldatei verstehen
- Thread mittels Klasse und `operator()` realisieren
- Atomare vs. nicht atomare Operationen in `C++`

5. So, jetzt ist noch der letzte Teil dieses kleinem Beispieles zu erledigen, nämlich den Fehler auszubessern. Das kannst du ganz gut alleine erledigen, indem du dafür sorgst, dass alle kritischen Abschnitte geschützt werden. Es gibt dafür die Möglichkeit den Mutex selbst zu locken, einen `lock_guard` oder einen `std::unique_lock` zu verwenden. Zeige, dass du jede Variante einsetzen kannst!

   Welche Variante ist wann vorzuziehen?

   **Übungszweck**

   - Locken mittels `mutex::lock()` und `mutex::unlock` sowie Locken mittels `lock_guard` und `unique_lock`.