Beispiel `03_sync1`

Dr. Günter Kolousek

7. Oktober 2020

1 Allgemeines

- In den ersten beiden Beispielen gibt es genaue Anweisungen zum Aufbau und der Durchführung eines Beispiels. Bei Bedarf nochmals durchlesen!
- Trotzedem hier noch zwei Erinnerungen:
  - **Regelmäßig** Commits erzeugen!
  - **Backup** nicht vergessen!
- In diesem Sinne ist jetzt ein neues Verzeichnis `03_sync1` anzulegen.

2 Aufgabenstellung

Dieses Beispiel hat den Sinn auf die verschiedenen Synchronisationsprobleme hinzuweisen. Der Name des zu erstellenden Programmes soll `account` heißen.
Los geht's!

3 Anleitung

1. Schreibe eine Klasse `Account` (siehe nächster Absatz), 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 dazu 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 sollen für die Klasse `Account` alle Methoden (member functions) in `account.cpp` erstellt werden.

Verwende für die Headerdatei dieses Mal einen Guard (und nicht `#pragma once`), da wir auch dies können wollen.

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). Beispielsweise könnte das Konto mit 15€ befüllt und danach zuerst 6€ abgebucht werden und anschließend wird versucht 10€ abzubuchen. Gib den jeweiligen Erfolg und den Kontostand am Schluss aus.

2. Schreibe jetzt das Programm so um, dass am Anfang 1€ in das Konto eingefügt 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?

3. Wahrscheinlich schon, aber das heißt nicht unbedingt, dass der Code fehlerfrei ist! Weiter zum nächsten Punkt!

4. 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? Starte notfalls das Programm **mehrmals**!

5. Jetzt sollte es klar sein, dass `balance -= amount` keine atomare Operation ist und auch die Abfrage auf `balance` schon alleine nicht thread-safe ist. Es handelt sich also um einen kritischen Abschnitt, der mit einem `lock_guard` geschützt werden will.

   Teste nochmals!

   Jetzt sollte dies soweit funktionieren. Gut so.

6. Schreibe jetzt eine eigene Klasse `Depositer` (im Modul `account`), die beim Initialisieren 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?

   Um eine Instanz als Thread starten zu können, muss die Klasse einen überladenen Operator `operator()()` haben, in der die Schleife abgearbeitet wird und die Methode `deposit()` aufgerufen wird.

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?

Ok, dann schreibe deine `deposit` Methode einmal um und zwar von:

```
balance += amount;
```

in:

```
balance = balance + amount;
```

Das ist klarerweise das Gleiche! Um allerdings das Problem zu zeigen, ändere wie folgt ab:

```
int tmp{balance};
std::this_thread::sleep_for(10ms);
balance = tmp + amount;
```

Im single-threaded Fall noch immer äquivalent bzgl. des Endergebnisses, aber im multi-threaded Fall…

Es sollte klar sein, dass es auch ohne eingebaute Verzögerung falsch wäre! `balance += amount` ist nun einmal **keine** atomare Operation.

Ok, was ist also zu tun? Wieder eine `lock_guard` einbauen. Allerdings werden wir einen `unique_lock` verwenden, wohl wissend, dass es nicht optimal ist, aber wir wollen auch diesen einmal zum Einsatz bringen und üben.

Damit kann das Schlafen auch wieder entfernt werden und auf den kompakten Ausdruck `balance += amount` zurückgeändert werden.

7. So, jetzt werden wir unser Beispiel wieder um eine kleine Benutzerschnittstelle erweitern. Der Anfangswert des Kontos und die Anzahl der 1€-Stücke, die auf das Konto von jedem Thread aufgebucht werden sollen, sollen mittels der Benutzerschnittstelle konfiguriert werden können. Das soll folgendermaßen aussehen:

```
$ account -h
Account app
Usage: account [OPTIONS] balance

Positionals:
  balance INT REQUIRED        Initial balance

Options:
  -h,--help                   Print this help message and exit
  -d,--deposits INT=5         Count of deposits

$ account --help
```

```
Account app
Usage: account [OPTIONS] balance

Positionals:
  balance INT REQUIRED        Initial balance

Options:
  -h,--help                   Print this help message and exit
  -d,--deposits INT=5         Count of deposits

$ account
balance is required
Run with --help for more information.
$ account 3a
Could not convert: balance = 3a
Run with --help for more information.
$ account 0
10
$ account 3
13
$ account -1
9
$ account -d 3 5
11
$ account --deposits=3 5
11
```

Natürlich können wir das programmieren, aber es ist sehr monoton und fehleranfällig. Besser man nimmt sich eine Library, die einem bei dieser Arbeit unterstützt.

Wir greifen auf die header-only Bibliothek `CLI11` zurück, für die ich eine entsprechende Datei `CLI11.hpp` zur Verfügung stelle. Studiere einmal die Homepage `https://github.com/CLIUtils/CLI11` und die Dokumentationsseite `https://cliutils.github.io/CLI11/book/`!

Im übernächsten Punkt werde ich Unterstützung bieten!

8. Bevor wir uns an das TUI wagen, werden wir noch ein kleines Refactoring durchführen: Derzeit ist die Anzahl der "Deposits" mit 5 hart kodiert in der Klasse `Depositer`. Jetzt soll diese Anzahl im Konstruktor der Klasse mitgegeben werden können.

9. Kopiere zuerst die Datei `CLI11.hpp` in dein Verzeichnis `include` und erweitere deine Funktion `main` um die folgenden Codezeilen:

```
CLI::App app("Account app");
```

```
int balance{0};
app.add_option("balance", balance, "Initial balance")->required();

int deposits{5};
app.add_option("-d,--deposits", deposits, "Count of deposits", true);

CLI11_PARSE(app, argc, argv);
```

Danach kannst du auf die Werte in den Variablen `balance` und `deposits` zugreifen. Im Fehlerfall wird der Prozess mit einem Exit-Code von `CLI11` abgebrochen. Willst du einen eigenen Exit-Code zurückliefern oder das Parsen der Kommandozeilenargumente innerhalb einer Funktion ausführen, dann musst du wissen was das Macro so tut (kann prinzipiell in der Dokumentation nachgesehen werden):

```
try {
    app.parse(argc, argv);
} catch (const CLI::ParseError &e) {
    return app.exit(e);
}
```

Ok, wir benötigen es jetzt aber nicht, also übersetze und teste!

In Zukunft musst du selber in der Lage sein, solche einfachen Benutzerschnittstellen (mittels `CLI11`) implementieren zu können.

4 Übungszweck dieses Beispiels

- Modul erstellen und Header Guards einsetzen
- Threads mit Lambdaausdruck realisieren.
- `this_thread::yield()` kennenlernen
- `using namespace std::literals;` kennenlernen
- `operator()()` üben
- Race conditions verstehen
- Initializerliste üben
- Implementierung in Headerdatei vs. in Quelldatei verstehen
- Atomare vs. nicht atomare Operationen in `C++`
- Locken mittels `mutex::lock()` und `mutex::unlock` sowie Locken mittels `lock_guard` und `unique_lock`.

- einfache textbasierte Kommandozeilen-basierten Programme mit `CLI11`
- Verwenden einer header-only Library durch Kopieren in `include`