Beispiel `06_future`

Dr. Günter Kolousek

26. November 2020

1 Allgemeines

- Im ersten Beispiel gibt es genaue Anweisungen zum Aufbau und der Durchführung eines Beispiels. Bei Bedarf nochmals durchlesen!
- Trotzdem hier noch zwei Erinnerungen:
  - **Regelmäßig** Commits erzeugen!
  - **Backup** nicht vergessen!
- In diesem Sinne ist jetzt ein neues Verzeichnis `06_future` anzulegen.

2 Aufgabenstellung

Aufgabe ist es, ein einfaches Programm zum Faktorisieren großer Zahlen zu schreiben. Los geht's!

3 Anleitung

1. Schreibe ein Programm `factoring`, das eine variable Anzahl an Zahlen per Kommandozeile als Argument erhält und diese vorerst nur auf der Standardausgabe ausgibt. Weiters soll dieses Programm wieder über eine vernünftige Benutzerschnittstelle verfügen, die Benutzerfehler erkennt und über eine "usage"-Ausgabe verfügt (auch mit `--help`).

   Da es sich um das Faktorisieren von großen Zahlen handeln soll, ist eine separate Bibliothek notwendig, die beliebig große Zahlen zur Verfügung stellt. Verwende dazu die bereitgestellte header-only Library `InfInt`, die ihre Funktionalität ausschließlich in der Headerdatei `InfInt.h` zur Verfügung stellt.

   Die "Beschreibung" für diese Bibliothek befindet sich in der bereitgestellten Datei `InfIntReadMe.txt` und dem Source-Code `;-)`.

Ein Problem ist, dass `CLI11` nicht direkt mit `InfInt` umgehen kann, aber wir können folgendermaßen vorgehen:

a) Lege einen `vector<string>` an, der direkt für `CLI11` zur Aufnahme einer variablen Anzahl an Argumenten verwendet werden kann.

b) Beim Anlegen eines Argumentes (`add_option`, aber eben ohne Verwendung von `-` bzw. `--` beim Namen), kann die Methode `check` aufgerufen werden, die sich ein Callable erwartet. Dieses Callable erwartet sich als Argument einen String (oder besser eine konstante Referenz auf einen String). Dieses Callable soll einen leeren String zurückliefern, wenn das Argument (oder der Teil davon, da ja ein `vector` verwendet wird) gültig ist und anderenfalls eine Fehlermeldung.

   Ungültig ist ein Argument, wenn es ein (oder mehrere) Zeichen enthält, die nicht die Ziffern 0 bis 9 darstellen. Dies kann mit der Methode `find_first_not_of` erreicht werden (siehe cppreference).

c) Hat die Eingabe funktioniert, dann kann ein `vector<InfInt>` angelegt werden und manuell aus dem `vector<string>` befüllt werden.

Das Programm sollte einmal bis jetzt so funktionieren:

```
$ factoring 1234567890 12345678901234567890
1234567890
12345678901234567890
$ factoring 1234567890 12345678901234567890 1x
number: 1x contains not numeric character
Run with --help for more information
$ factoring --help
Factor numbers
Usage: factoring [OPTIONS] number...

Positionals:
  number TEXT ... REQUIRED    numbers to factor

Options:
  -h,--help                   Print this help message and exit
  -a,--async                  async
```

2. Erweitere jetzt dein Programm insoferne, dass anstatt der bisherigen Ausgabe jetzt die Primfaktoren berechnet werden und diese in der folgenden Form ausgegeben werden:

```
$ factoring 1234567890 12345678901234567890
1234567890: 2 3 3 5 3607 3803
12345678901234567890: 2 3 3 5 101 3541 3607 3803 27961
$
```

Um die Primfaktoren zu berechnen, habe ich eine eigene Headerdatei mit dem Namen `calc_factors.h` zur Verfügung gestellt.

3. In weiterer Folge ist es die Idee die Berechnung der Primfaktoren asynchron vorzunehmen.

   Erweitere deshalb das Programm so, dass die eigentliche Berechnung **aller** Zahlen über die Funktion `async` angestoßen wird und die Ergebnisse **danach** ausgegeben werden.

   Tipp: Lege einen Vektor von `future` Instanzen an!

   Die Ausgabe soll sich im Vergleich zum vorherigen Punkt nicht geändert haben.

4. Jetzt wird die Ausgabe der Ergebnisse in eine eigene Funktion refaktorisiert, die den Vektor der Zahlen und den Vektor der Ergebnisse per Referenz bekommt. Danach wird diese Funktion als eigener Thread gestartet.

   Dieser Schritt dient als Vorbereitung auf die kommenden Änderungen.

   Die Ausgabe soll noch immer gleich aussehen.

5. In weiterer Folge wollen wir einen weiteren Thread starten, dessen Aufgabe es sein soll, die Faktorisierung zu überprüfen, indem die ursprüngliche Zahl mit dem Produkt der Faktoren verglichen wird.

   Ist die Faktorisierung für eine Zahl nicht erfolgreich, dann ist eine Fehlermeldung auf `stderr` auszugeben.

   Wichtig hierbei ist, dass jetzt zwei Threads auf ein Future zugreifen wollen und dies nicht thread-safe ist. Daher ist der Code so umzubauen, dass anstatt eines `future` ein `shared_future` verwendet wird. Beachte, dass jetzt deine Funktion zum Ausdrucken ebenfalls umzuändern ist, da ein `shared_future` kopiert werden muss, damit jeder Thread seine eigene Kopie des `shared_future` hat.

6. Jetzt soll noch die benötigte Zeit zum Faktorisieren ermittelt werden.

   Im Großen und Ganzen funktioniert das so, dass zuerst die Startzeit ermittelt wird, die Endezeit nachdem alle Zahlen faktorisiert worden sind und danach die Differenz ermittelt wird (eh klar).

```
// measuring time part 1
auto start = chrono::system_clock::now();

// factorizing...

// measuring time part 2
auto duration = chrono::duration_cast<chrono::milliseconds>
    (std::chrono::system_clock::now() - start);
cout << "Time elapsed used for factoring: " << duration.count() << "ms";
```

Bitte beachte, dass es sich hier lediglich um die prinzipielle Vorgangsweise handelt. Im konkreten Fall können durchaus noch Änderungen notwendig sein. Messe aber nicht wenn sich die Threads beendet haben, sondern wenn die alle Ergebnisse vorliegen!

Die Ausgabe sollte jetzt folgendermaßen aussehen:

```
$ factoring 1234567890 12345678901234567890 123456789012345678901234567890
1234567890: 2 3 3 5 3607 3803
12345678901234567890: 2 3 3 5 101 3541 3607 3803 27961
123456789012345678901234567890: 2 3 3 3 5 7 13 31 37 211 241 2161 3607 3803 29061
Time elapsed used for factoring: 253ms
$
```

7. Nachdem wir jetzt in der Lage sind, die Zeit zu messen, können wir uns das `async` Verhalten genauer ansehen.

   Erweitere die Benutzerschnittstelle um eine Option `--async`. Wird diese Option vom Benutzer angegeben, dann sollen die `async` - Aufrufe wirklich als eigener Thread gestartet werden.

   Die Ausgabe könnte jetzt folgendermaßen aussehen:

   ```
   $ factoring --async 1234567890 12345678901234567890 123456789012345678901234567890
   1234567890: 2 3 3 5 3607 3803
   12345678901234567890: 2 3 3 5 101 3541 3607 3803 27961
   123456789012345678901234567890: 2 3 3 3 5 7 13 31 37 211 241 2161 3607 3803 29061
   Time elapsed used for factoring: 131ms
   $
   ```

   Untersuche jetzt ein bischen das zeitliche Verhalten deines Programmes, also starte es einmal mit einer Zahl, dann mit zwei gleichen Zahlen, dann mit drei gleichen Zahlen,… Irgendwelche Beobachtungen?

   ```
   $ factoring --async 123456789012345678901234567890
   123456789012345678901234567890: 2 3 3 3 5 7 13 31 37 211 241 2161 3607 3803 29061
   Time elapsed used for factoring: 101ms
   $ factoring --async 123456789012345678901234567890 123456789012345678901234567890
   123456789012345678901234567890: 2 3 3 3 5 7 13 31 37 211 241 2161 3607 3803 29061
   123456789012345678901234567890: 2 3 3 3 5 7 13 31 37 211 241 2161 3607 3803 29061
   Time elapsed used for factoring: 115ms
   $
   ```

   Starte auch das Programm mehrmals mit den gleichen Werten…

4 Übungszweck

- Kommandozeilenschnittstelle üben: Vertiefen der Verwendung von `CLI11`
- Einsatz von `async` und `Future` üben.
- Üben von Threads mit Übergabe der Parameter per Referenz
- Erkennen, dass `future` Instanzen nicht kopiert werden können (da es keinen Sinn macht).
- `cerr` verwenden
- Notwendigkeit von `shared_future` erkennen und einsetzen
- Zeit messen in `C++`
- Zeitliches Verhalten erkunden