Beispiel `02_threads`

Dr. Günter Kolousek

30. August 2020

1 Allgemeines

- 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** nicht vergessen!
- Verwende das bereitgestellte Archiv `template.tar.gz` zum Erstellen eines Meson-Projektes. Es enthält die notwendigen Anpassungen zur Verwendung von Threads. **Lösche** Alles (Dateien, Verzeichnisse, Inhalt von `meson.build`,…), das nicht benötigt wird!
  - Wichtig: Die `default_options` in `meson.build` sind so zu belassen!
- Die Anpassungen der Datei `.hgignore` bzw. `.gitignor` sollten schon erledigt sein, sodass das Verzeichnis `build` nicht versioniert wird.
- In diesem Sinne ist jetzt ein neues Verzeichnis `02_threads` anzulegen.

2 Aufgabenstellung

Aufgabe ist es, eine Simulation eines Auto Wettrennens in einem Programm `contest` zu schreiben.
Dieses Beispiel dient eigentlich nur der Wiederholung und dem Auffrischen des Wissens über Threads… allerdings in `C++`!
Los geht's!

3 Anleitung

1. Schreibe eine Funktion für eine Autotype deiner Wahl (z.B. `lada_taiga`), das ein Auto (eben eines Lada Taiga) auf einer virtuellen Rennstrecke Runden fahren lässt. Du kannst/sollst natürlich deine eigene Automarke verwenden.

   Aufgabe ist es, dass das Auto permanent Runden fährt und nach jeder Runde eine entsprechende Ausgabe als eigene Zeile (Rundennummer Auto) auf der Konsole ausgibt (z.B. nach der ersten Runde '1 Lada Taiga', nach der zweiten Runde '2 Lada Taiga', usw.). Die Zeit einer Runde kann vorerst durchaus konstant sein, wie zum Beispiel jeweils 1 Sekunde.

   Für eine Sekunde warten geht sehr leicht, vorausgesetzt es werden alle notwendigen Headerdateien inkludiert:

   ```
   // do not forget about using std::literals;
   this_thread::sleep_for(1s);
   ```

   Diese Funktion soll in `main` als Thread gestartet werden.

   Beendet wird das Programm durch Abbruch.

   Teste dein Programm in der **Konsole** (und nicht nur in einer IDE), da wir später mehrere Prozesse mit verschiedenen, sich ändernden Benutzereingaben gestartet werden und deren Ausgaben "gleichzeitig" beobachtet werden sollen. Das funktioniert **viel** einfacher und übersichtlicher in mehreren Terminalfensterm als in einer IDE!

2. Schreibe eine Klasse `Car`, die ebenfalls als Thread verwendet werden kann, die die gleiche Funktionalität aufweist wie im vorhergehenden Punkt, jedoch im Konstruktor die Automarke mitgegeben werden kann, wie z.B. "Opel Manta". Um die Instanz direkt im Konstruktor von `thread` übergeben zu können, ist `operator()` zu implementieren:

   ```
   void operator()() {
       // ...
   }
   ```

   Eine Instanz dieser Klasse soll also in `main` als Thread gestartet werden.

   Teste wiederum!

   Was fällt dir auf? Das sieht vermutlich nicht hübsch aus, aber das liegt daran, dass mehrere Threads eine Ressource nutzen, nämlich die Ausgabe. Darum kümmern wir uns im nächsten Punkt!

3. Jetzt wollen wir einmal dieses optische Problem beheben. Ändere deine Ausgabeanweisungen so um, dass diese nicht aus zwei Ausgaben hintereinander (eine ganze Zahl und ein String) tätigen, sondern nur eine Ausgabe vornehmen.

   Dazu muss aus den beiden Informationen ein String gebildet werden, der danach ausgegeben werden kann.

Mittels `to_string(x)` kannst du `x` in einen String wandeln, wobei es sich dem Typ von `x` um einen beliebigen arithmetischen Datentyp, bei uns konkret um einen `int`, handeln kann.

Teste!

Wenn du dich genau an diese Angaben gehalten, hast wird auch noch hie und da zu Problemen bei dem Zeilenumbruch kommen, nicht wahr? Kannst du das selber lösen, dann gehe direkt zum übernächsten Punkt, ansonsten mache vertrauensvoll beim nächsten Punkt weiter.

Passe auf, dieses Vorgehen wird wahrscheinlich funktionieren, nur ist es keine Garantie für ein Funktionieren, da die mehrfache Verwendung des überladenen Operator `<<` nicht thread-safe ist und wir eine Wartezeit in jedem Schleifendurchgang eingefügt haben. Allerdings werden wir uns damit im Moment nicht weiter beschäftigen, sondern erst in einem späteren Beispiel.

4. Ok, analysieren wir einmal. Auch das liegt daran, dass der Operator `<<` mehrmals verwendet wird (beim Einsatz von `endl`). Wie kann man das lösen? Ersetze einfach das `endl` mit einem gezieltem Einsatz von `\n` und `flush`. Erledigt (aber wie im vorhergehenden Punkt erwähnt nicht ganz richtig)!

   Bedenke, dass du nur dann `endl` in perforamance-kritischen Programmen einsetzen sollst, wenn du ein implizites `flush` auch wirklich benötigst. Ansonsten tut es ein gutes, altes `\n` auch!!!

5. Erweitere die beiden Auto-Threads: jede Runde soll eine zufällige Rundenzeit in der Länge eines Intervalles von 1 (inklusive) bis 10 (exklusive) Sekunden aufweisen.

   Verwende dazu folgenden Code und schaue dir dazu die Dokumentation an:

   ```
   #include <random>

   std::random_device rd;
   std::mt19937 gen{rd()};
   std::uniform_real_distribution<> dis{1, 10};

   cout << dis(gen) << endl;
   ```

   Beachte, dass das Schlafen jetzt in Sekunden so einfach nicht mehr funktioniert. Verwende deshalb den ermittelten Wert und rechne diesen in Millisekunden um. Dann kannst du `chrono::milliseconds{x}` verwenden. D.h. Die Genauigkeit für das Warten soll in Millisekunden sein.

6. Die Ausgabe soll um die Ausgabe der Rundenzeit erweitert werden, z.B.: '1 Opel-Manta: 4.45', wobei maximal 2 Nachkommastellen ausgegeben werden.

   Hier besteht die Möglichkeit die Zahl so umzurechnen, dass es bis zu 2 Nachkommastellen gibt oder (besser) es sind die Möglichkeiten eines `ostream` und des Manipulators `setprecision` (Header `<iomanip>`) auszuschöpfen.

Allerdings kann `cout` (vom Typ eines `ostream`) nicht verwendet werden, da es eine gemeinsame Ressource (globale Variable) ist.

Deshalb ist eine lokale Variable zu verwenden, nämlich vom Typ `ostringstream` (Header `<sstream>`), der ein Ausgabestream ist, dessen Backend ein String ist:

```
ostringstream buf;
buf << 42 << "foo" << endl;
string str = buf.str();  // -> "42foo"
buf.str("");  // reset buf
```

Die Ausgabe könnte bis jetzt ungefähr folgendermaßen aussehen:

```
0 Lada Taiga 4.17s
0 Opel Manta 4.35s
1 Opel Manta 2.47s
1 Lada Taiga 4.27s
2 Lada Taiga 6.95s
2 Opel Manta 9.58s
3 Lada Taiga 1.03s
3 Opel Manta 1.33s
...
```

7. Erweitere das Programm so, dass jetzt jedes Auto nur mehr 10 Runden fährt, die Gesamtzeit ermittelt wird und das Gesamtergebnis vom Hauptthread am Ende ausgegeben wird (also jeweilige Gesamtzeit und Sieger).

   Dazu muss offensichtlich jeder Thread seine jeweilige Gesamtzeit ermitteln und das Ergebnis muss an den Hauptthread zurückgegeben werden.

   Bei der Klasse ist das relativ einfach: Einfach eine Methode `get_total_time` schreiben, die die Gesamtzeit zurückliefert, aber wie ist das bei dem Thread auf Basis der Funktion zu tun? Denke einmal kurz nach und schaue erst dann zum nächsten Absatz. Schummeln ist unsinnig, denn es bringt (dir) nichts. Also **denke** kurz nach.

   Ok, es ist ganz einfach: gib das Gesamtergebnis einfach als Returnwert der Funktion zurück.

   Gut, dann kannst du zum nächsten Punkt weitergehen. Wenn du allerdings meinst, dass das ziemlicher Schwachsinn ist, dann liegst du richtig. Es muss die Funktion um eine lvalue-Referenz als Parameter ergänzt werden. Schaue dir die diesbezüglichen Folien gegebenenfalls nochmals an (achte auch auf `std::ref`!).

   Vielleicht überlegst du dir auch noch, ob das überhaupt sicher ist, da es sich bei dem Referenzparameter um eine gemeinsame Ressource handelt wie auch bei `cout`. Ja, das ist schon richtig, allerdings greift der Hauptthread nur darauf (lesend) zu, wenn der andere Thread sich schon beendet hat. Das gleiche Argument kann auch für die Lösung mit der Klasse vorgebracht werden.

Ok, dann wirst du vermutlich eine Ausgabe erhalten, die so ähnlich aussieht wie folgt:

```
$ race
1 Lada Taiga 2.54s
1 Opel Manta 6.31s
2 Lada Taiga 4s
3 Lada Taiga 3.39s
2 Opel Manta 6.78s
4 Lada Taiga 5.66s
3 Opel Manta 2.71s
4 Opel Manta 6.13s
5 Lada Taiga 6.47s
5 Opel Manta 2.93s
6 Lada Taiga 7.29s
6 Opel Manta 6.22s
7 Opel Manta 4.98s
8 Opel Manta 2.07s
7 Lada Taiga 9.8s
8 Lada Taiga 5.17s
9 Opel Manta 7.92s
9 Lada Taiga 2.13s
10 Lada Taiga 6.48s
10 Opel Manta 9.73s
Sieger ist: Opel Manta mit 0s
Verlierer ist: Lada Taiga mit 52.918s
```

Jetzt stellt sich natürlich die berechtigte Frage warum das Ergebnis von meinem superschnellen Open Manta 0s ist… Diesem "kleinem" Problem werden wir uns gleich im nächstem Punkt widmen.

8. So, jetzt widmen wir uns dem Problem, dass als Gesamtzeit für die Klassen-basierte Thread-Lösung nur 0 ermittelt worden ist. Gut, probieren wir es wieder mit Nachdenken…

   Problem erkannt?

   - Ja? Gut! Die Behebung des Problems ist allerdings wahrscheinlich nicht so klar.
   - Nein? Vielleicht hilft ein kleiner Tipp? Das Problem ist in dieser Übung schon irgendwie einmal vorgekommen und wurde auch schon einmal gelöst.

   Ok, hier die Auflösung: Die Instanz der Autoklasse wird als Kopie an das Thread-Objekt übergeben!

   Hierfür gibt es zwei Lösungen:

- Die einfache Lösung ist, dass du die Instanz als Referenz übergibst. Dazu ist wiederum `std::ref` zu einzusetzen:

  ```
  thread opel_thread{ref(opel_manta)};
  ```

- Die zweite (kompliziertere) Lösung ist, dass ein Pointer auf die entsprechende member function (C++ Nomenklatur für Methode) und eine Referenz auf das eigentliche Objekt zu übergeben ist:

  ```
  thread opel_thread{&Car::operator(), ref(opel_manta)};
  ```

  Klarerweise wirst du die erste Lösung verwenden, aber hier zeige ich dir eben wie du einen Pointer auf eine Methode in C++ anschreiben kannst und außerdem wird dir bewusst, dass eine Methode nichts anderes als eine Objekt-gebundene Funktion ist.

Damit sollte das Ergebnis einer Simulation ungefähr folgendermaßen aussehen:

```
$ race
1 Opel Manta 3.63s
1 Lada Taiga 7.3s
2 Opel Manta 4.04s
2 Lada Taiga 3.36s
3 Opel Manta 6.07s
3 Lada Taiga 6.83s
4 Opel Manta 4.08s
5 Opel Manta 2.2s
6 Opel Manta 2.21s
4 Lada Taiga 6.3s
7 Opel Manta 3.96s
5 Lada Taiga 6.77s
8 Opel Manta 7.14s
6 Lada Taiga 4.21s
7 Lada Taiga 5.07s
9 Opel Manta 8.65s
10 Opel Manta 1.98s
8 Lada Taiga 4.41s
9 Lada Taiga 1.11s
10 Lada Taiga 9.24s
Sieger ist: Opel Manta mit 43.975s
Verlierer ist: Lada Taiga mit 54.579s
```

9. Implementiere jetzt eine einfache text-basierte Benutzerschnittstelle mittels Kommandozeilenargumeten, die folgendermaßen funktioniert:

```
$ contest -h
Usage: contest [-h | --help | LAPS]
```

```
$ contest --help
Usage: contest [-h | --help | LAPS]
$ contest 0
Out of range (1 <= LAP'S < 16): 0
Run with --help for more information.
$ contest a
Could not convert: a
Run with --help for more information.
$ contest 3a
Could not convert: 3a
Run with --help for more information
```

Bei fehlerhafter Angabe eines Kommandozeilenargumentes ist die entsprechende Fehlermeldung auf stderr auszugeben und das Programm mit dem Exit-Code 1 zu beenden.

Wird nur eine Hilfe angefordert, dann ist die "Usage"-Meldung auf stdout auszugeben und das Programm mit dem Exit-Code 0 zu beenden.

Um diese gewünschte Funktionalität zu erreichen entwickle eine Funktion `void help()` und eine Funktion `void error(string msg="")` Für `error` gilt, dass die übergebene Nachricht nur ausgegeben wird, wenn diese ungleich dem Leerstring ist, ansonsten wird nur der Hinweis ausgegeben, dass das Programm mit `--help` aufgerufen werden soll.

Wird eine korrekter Wert angegeben, dann ist sind die angegebene Anzahl an Runden zu fahren. Wird kein Argument mitgegeben, dann ist für die Rundenanzahl wiederum der Wert 10 zu nehmen.

Und jetzt noch ein paar Tipps bzw. Richtlinien (d.h. sind einzuhalten):

- `int main(int argc, char* argv[])` ist die korrekte Signatur für `main`.
- Für die Umwandlung eines Strings ist die Funktion `stoi` zu verwenden.
  - `stoi` liefert Exceptions zurück, z.B. bei Übergabe von `a`.
  - `stoi` liefert keine Exception bei Übergabe von `3a`! Hier musst du den Parameter `pos` verwenden!
- Die korrekte Rundenanzahl hat als Argument per-value an den Thread übergeben zu werden.

4 Übungszweck dieses Beispiels

- Meson vertiefen
- Wiederholung von Threads und Verständnis für Threads vertiefen.

- Starten eines Threads auf Basis einer Funktion, `join`.
- Blockieren eines Threads für eine bestimmte Zeitspanne mittels `sleep_for`.
- Zeitliterale mittels `using std::literals;` aktivieren
- Thread als Klasse und `operator()` realisieren.
- Erkennen, dass es zu Problemen kommen kann, wenn mehrere Threads auf eine Ressource zugreifen.
- `to_string` kennenlernen.
- Ersetzen von `endl` durch `\n` und `flush`
- Zufallszahlen ermitteln.
- `chrono::milliseconds` und `static_cast` verwenden.
- Manipulatoren üben
- `stringstream` kennenlernen
- Ausgabe eines Ergebnisses aus einer Funktion mittels Referenzparameter.
- Übergebenen eines Objektes an einen Thread per-value und per-reference.
- Einfache Benutzerschnittstelle per Kommandozeile