1 Projekt strukturieren und aufsetzen

In dieser Aufgabe sollen Sie explizit nicht das Beispiel-Projekt verwenden, sondern von Grund auf ein neues Projekt aufsetzen. Gliedern Sie dies auch in das `workspace` Repo ein. Damit dieses Sub-Projekt auch in das Hauptprojekt aufgenommen wird, müssen Sie im Top-Level `CMakeLists.txt` einen entsprechenden weiteren `add_subdirectory()` Eintrag erstellen.

Wir verwenden in dieser Übung kein `cpplint`. Zudem definieren wir das CMake Target für die Applikation und die Tests komplett unabhängig und deshalb ohne eigene Basis-Library.

1.1 Aufgabe

- Erstellen Sie einen Unterordner `build_system` im `workspace` Repo.
- Erstellen Sie eine `CMakeLists.txt`-Datei in welcher Sie das Projekt definieren.
- Strukturieren Sie ihr Projekt in weiteren Unterordner. Diese sollen bereits jetzt Dateien für ihr Hauptprogramm und Unittests vorsehen.
- Folgende Dateien sollen im Projekt enthalten sein:
  - `main.cpp`
  - `student.h` und `student.cpp`
  - `module.h` und `module.cpp`
- Definieren Sie ein Executable-Target mit dem Namen `build_system_app`. Tragen Sie die dazugehörigen Source-Dateien einzeln ein.
- Definieren Sie ihren Unterordner als Include-Ordner.
- Setzten Sie die bereits definierten Compiler-Flags in der Variable `cxxflags` in ihrem Executable-Target.

Bevor Sie das Projekt kompilieren können, müssen Sie zuerst die Implementation der nächsten Aufgabe erledigen.

2 Implementation

Nun sollen Sie die vorbereiteten Dateien mit Inhalt füllen. Die Aufgabe ist, die öffentlichen Klassen `Student` und `Module` zu implementieren und in einem Executable zu verwenden. Das Executable soll folgenden Output zur Runtime generieren:

```
1 Module: prcpp, ECTS: 3
2 Module: sysad, ECTS: 3
```

Die Klasse `Student` soll einen Default-Konstruktor und folgende Methoden anbieten:

```
size_t GetModuleCount();
Module GetModule(size_t index);
bool AddModule(Module new_module);
bool RemoveModule(size_t index);
```

Die interne Verwaltung der Module soll mittels statischem Array implementiert werden:

```
static constexpr size_t kMaxModules = 4;
Module modules_[kMaxModules];
```

Die Klasse `Module` soll eine Member-Variable für die Anzahl ECTS-Punkte und eine für den Modul-Namen besitzen. Verwenden sie `std::string` für den Namen.

2.1 Aufgabe

- Definieren und Implementieren Sie die Klassen `Student` und `Module`.
- Verwenden Sie `#pragma once` in allen Header-Dateien als erste Instruktion um Mehrfach-Inkludierung des Präprozessors zu verhindern.
- Verwenden Sie `#include <cassert>` um bei Bedarf Assertions zu schreiben.
- Implementiere Sie `main.cpp`. Nun können Sie die Applikation kompilieren und starten.

3 Unit-Tests

Im letzten Teil soll mittels dem verfügbaren Test-Framework Catch2 einige Tests für die Klasse `Student` geschrieben werden. Dazu müssen Sie nur den Header `catch2/catch.hpp` in ihrer Test-Datei inkludieren. Nennen Sie diese Datei `student_test.cpp`. Zudem müssen Sie einen Einstiegspunkt für das Test-Executable definieren. Dies machen Sie mittels einer weiteren `main.cpp` und folgendem Inhalt:

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
```

3.1 Aufgabe

- Erstellen Sie die Datei `main.cpp` im dafür vorbereiteten Unterordner.
- Erstellen Sie die Datei `student_test.cpp` und implementieren Sie entsprechende Tests.
- Ergänzen Sie die Datei `CmakeLists.txt` um das neue Testprojekt. Dies soll ein weiteres Executable-Target sein.

4 Memory-Checking mit Valgrind

In C++ operieren Sie auf der echten Maschine ihres Systems und haben deshalb weniger Sicherheits-Mechanismen als z.B. in einer JVM. Ein übliches Problem ist falscher Code, welcher auf Speicher operiert, welcher entweder nicht initialisiert wurde oder gar nicht dem aktuellen Prozess gehört. Im ersten Fall erzeugt dies unter Umständen ein unerwartetes/falsches Verhalten der Applikation, im zweiten Fall führt dies zum Absturz mit dem Fehler `SIGSEGV` oder eben *Segmentation-Fault*.

Um solche Probleme zu analysieren und zu debuggen gibt es unter Linux das freie Tool `valgrind`. Im Standard-Modus analysiert es die Allozierung und Verwendung allen Speichers ihrer Applikation und meldet eventuelle Probleme. Dies wird intern so implementiert, dass `valgrind` denn Allocator von der C++-Runtime austauscht und somit jede Memory-Allozierung selber durchführen und überwachen kann.

Sie können `valgrind` entweder über die Konsole laufen lassen oder direkt über CLion. Konfigurieren müssen Sie dazu nichts. Siehe: CLion - Valgrind Memcheck

Versuchen Sie z.B. die Methode `Student::RemoveModule` so zu manipulieren, dass der Loop auf mehr Zellen operiert als er eigentlich sollte. Je mehr er vom eigentlichen Memory abweicht, desto wahrscheinlicher erhalten Sie nicht nur Warnings sondern einen *Segfault*.