05_calc_awards: Ermittlung des Schulerfolges

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

2 Aufgabenstellung

Schreibe ein C++ Programm `awards`, das für eine Liste von Schülern und deren Noten berechnet, welchen Schulerfolg der jeweilige Schüler erreicht hat. Die möglichen Schulerfolge sind "nicht bestanden", "bestanden", "mit gutem Erfolg bestanden", "mit ausgezeichnetem Erfolg bestanden".

Die Regeln hierfür sind in aufsteigender Reihenfolge:

nicht bestanden mindestens ein *Nicht genügend*

bestanden kein *Nicht genügend*

mit gutem Erfolg bestanden kein *Genügend* und Notendurchschnitt ≤ 2

mit ausgezeichnetem Erfolg bestanden kein *Genügend* und Notendurchschnitt ≤ 1.5

Die Noten sind in einer CSV-Datei gespeichert, die folgenden Aufbau hat (Felder jeweils durch `,` getrennt):

- 1.Zeile: Headerzeile,
  - die das Namensfeld enthält
  - danach beliebige Anzahl an Gegenstandsbezeichnungen
- Weitere Zeilen sind die eigentlichen Datensätze

Anzahl der Felder muss in jedem Datensatz mit der Headerzeile übereinstimmen. Das ist zu überprüfen!

Die folgende Datei `marks.txt` stelle ich zur Verfügung:

```
name,POS,NVS,DBI,BWM,D,AM,GGP,NW,BESP
Maxi,1,1,1,1,1,1,1,1,1
Mini,1,1,1,1,1,1,1,5,1
Otto,1,1,1,1,1,1,1,4,1
Erna,1,2,1,2,1,2,1,2,1
Susi,1,3,1,1,3,1,3,1,1
Anna,1,1,1,3,3,3,3,3,1
Robi,1,1,1,1,1,1,1,1,1
```

Damit hat die Benutzerschnittstelle folgendermaßen zu funktionieren:

```
$ awards -h
Calculates the awards for the given students
Usage: awards [Options] [STDIN]

Positionals:
  STDIN TEXT                  stdin marker (must be '-')

Options:
  -h,--help                   Print this help message and exit
  -i,--infile TEXT            The file to be processed (if ommited: stdin
  -o,--outfile TEXT           The file to be written (if ommited: stdout)
$ awards -i marks.txt
name,POS,NVS,DBI,BWM,D,AM,GGP,NW,BESP,award
Maxi,1,1,1,1,1,1,1,1,1,mit ausgezeichnetem Erfolg bestanden
Mini,1,1,1,1,1,1,1,5,1,nicht bestanden
Otto,1,1,1,1,1,1,1,4,1,bestanden
Erna,1,2,1,2,1,2,1,2,1,mit ausgezeichnetem Erfolg bestanden
Susi,1,3,1,1,3,1,3,1,1,mit gutem Erfolg bestanden
Anna,1,1,1,3,3,3,3,3,1,bestanden
Robi,1,1,1,1,1,1,1,1,1,mit ausgezeichnetem Erfolg bestanden
$ awards -i marks.txt -o results.txt
$ cat results.txt
name,POS,NVS,DBI,BWM,D,AM,GGP,NW,BESP,award
Maxi,1,1,1,1,1,1,1,1,1,mit ausgezeichnetem Erfolg bestanden
Mini,1,1,1,1,1,1,1,5,1,nicht bestanden
Otto,1,1,1,1,1,1,1,4,1,bestanden
Erna,1,2,1,2,1,2,1,2,1,mit ausgezeichnetem Erfolg bestanden
Susi,1,3,1,1,3,1,3,1,1,mit gutem Erfolg bestanden
Anna,1,1,1,3,3,3,3,3,1,bestanden
Robi,1,1,1,1,1,1,1,1,1,mit ausgezeichnetem Erfolg bestanden
$ cat marks.txt | awards -
name,POS,NVS,DBI,BWM,D,AM,GGP,NW,BESP,award
Maxi,1,1,1,1,1,1,1,1,1,mit ausgezeichnetem Erfolg bestanden
Mini,1,1,1,1,1,1,1,5,1,nicht bestanden
Otto,1,1,1,1,1,1,1,4,1,bestanden
Erna,1,2,1,2,1,2,1,2,1,mit ausgezeichnetem Erfolg bestanden
Susi,1,3,1,1,3,1,3,1,1,mit gutem Erfolg bestanden
Anna,1,1,1,3,3,3,3,3,1,bestanden
Robi,1,1,1,1,1,1,1,1,1,mit ausgezeichnetem Erfolg bestanden
```

3 Anleitung

Schreibe ein Programm entsprechend der Aufgabenstellung.

1. Bevor wir zum "harten" Kodieren kommen, überdenken wir nochmals unsere derzeitige Build-Umgebung! Wie sieht es mit der Übersetzungszeit aus? Derzeit verwenden wir 3 header-only Bibliotheken: `doctest` wir nur für die Tests verwendet und `CLI11` eigentlich nur in der Datei `main.cpp` (oder einer Datei, die die Benutzerschnittstelle implementiert). Beide werden nur bedingt ins Gewicht fallen.

   Anders sieht es mit `fmt` aus. Diese wird in einem größeren Projekt unter Umständen in vielen Modulen verwendet werden. Das bedeudet, dass diese Headerdatei bei jeder Übersetzung inkludiert werden muss. Wenn wir uns das Einbinden von `fmt` nochmals vor Augen führen, dann sehen wird, dass wir ei-

ne `#define FMT_HEADER_ONLY` Präprozessordirektive angegeben hatten. Diese steuert offensichtlich das Verhalten der nachfolgenden `#include`-Direktiven. Weiters ist es offensichtlich, dass das Weglassen dieser Präprozessordirektive es möglich machen wird diese Bibliothek nicht nur im header-only Modus zu verwenden. Nicht im header-only Modus bedeutet, dass wir die Bibliothek zuerst übersetzen müssen und dann zu unserem Executable linken müssen. Gehe dazu folgendermaßen vor:

a) Gehe in das Projektverzeichnis von `fmt`. Dort lege ein Verzeichnis `build` an und wechsle in dieses. Dann ist dort ein beherztes `cmake ..` einzugeben. U.U. musst du `cmake` zuerst installieren, aber das ist deiner Lieblingsdistribution leicht mittels `sudo pamac -S cmake` durchzuführen. Danach folgt ein weiteres beherztes `make` (erkennst du die Ähnlichkeiten zu `meson`). Am Ende liegt im aktuellen Verzeichnis eine Datei `libfmt.a` vor.

b) Im nächsten Schritt wird die Bibliothek in `meson.build` eingebunden. Siehe dir dazu wieder den relevanten Abschnitt in `meson_tutorial` an. Beachte, dass die fertige Archivdatei von `fmt` den Namen `libfmt.a` hat, aber die Bibliothek nur unter dem Namen `fmt` ausgewählt wird (Präfix `lib` und Postfix `.a` werden implizit hinzugefügt, um zum Dateinamen zu kommen.

   Auch hier macht es durchaus Sinn, eine weitere Option mit dem Namen `fmt_build_dir` in `meson_options.txt` hinzuzufügen und in weiterer Folge in `meson.build` einzusetzen.

Teste, ob deine neue Konfiguration auch funktioniert (z.B. unter Verwendung einer Test-`fmt::print`-Anweisung).

2. Schreibe jetzt ein Modul `file_utilities`, das die folgenden Funktionen enthält:

```
vector<string> read_textfile(istream& file); // throws runtime_error
vector<string> read_textfile(const string filename); // throws runtime_error
void write_textfile(ostream& file, vector<string> lines); // throws runtime_error
void write_textfile(const string filename, vector<string> lines); // throws runtime_error
```

   Diese Funktionen sollen eine Textdatei zeilenweise lesen bzw. zeilenweise schreiben. D.h. die Funktion `getline()` kann wieder verwendet werden. Um ein korrektes Lesen oder Schreiben festzustellen, muss der Stream abgefragt werden. Dazu sind die Methoden `good()`, `eof()`, `fail()` und `bad()` geeignet. Es gibt auch die Möglichkeit, dass man Exceptions erhält, aber das muss separat aktiviert werden.

   - Warum gibt es jeweils zwei überladene Funktionen?
   - Denke daran, dass die eine Funktion die andere Funktion verwenden kann
   - Wenn ein Fehler in einer aufgerufenen Funktion diesen Fehler mittels Werfen einer Excpetion meldet, dann kann die aufrufende Funktion diese Abfangen und mittels der Funktion `throw_with_nested()` eine neue Exception werfen.
   - Das Abfangen einer Exception geschieht am besten per Referenz!
   - Beachte auch, dass `istream` anstatt `ifstream` und `ostream` anstatt `ofstream` verwendet wird. Warum wohl? Auch ein `cout` ist ein `ostream`...
   - Vergiss nicht auf das Schließen mittels `close()`. Man kann sich das zwar prinzipiell in bestimmten Situationen einsparen, wenn das Objekt wieder vom Stack verschwindet, da `close()` im Destruktor aufgerufen wird, aber...

   Auch wenn du es in diesem Beispiel nicht benötigst: Schaue dir auch die Methoden `clear()`, `seekg()`, `seekp()`, `tellg()`, `tellp()`, `flush()` an (die cppreference hat hierfür auch immer Beispielprogramme).

3. Nur mit dem Lesen von Zeilen ist es nicht getan. Diese müssen auch noch an einem Trennzeichen gesplittet werden. Dafür gibt es keine Funktionalität in der Standbibliothek, aber sehr viele Möglichkeiten wie dies (auch mit Hilfe der Standardbibliothek) realisiert werden kann.

Wir werden eine einfache Möglichkeit wählen, die auf Streams basiert. Als Streams haben wir schon die globalen Variablen `cin`, `cout` und `cerr`, sowie Typen `ifstream` und `ofstream` kennengelernt. Abgesehen von der Ein/Ausgabe auf die Konsole bzw. einer Datei gibt es auch noch die Möglichkeit einen String streammäßig zu beschreiben bzw. zu lesen.

Schreibe ein Modul `string_utilities`, das folgende Funktion enthält, die ich zur Gänze angebe, damit du diese "studieren" kannst:

```
vector<string> split(const string& s, char delimiter) {
    vector<string> tokens;
    string token;
    istringstream token_stream(s);
    while (getline(token_stream, token, delimiter)) {
       tokens.push_back(token);
    }
    return tokens;
}
```

Spätestens jetzt wäre es ein guter Zeitpunkt sich die Klasse `std::string` genauer anzusehen. Du wirst diese noch oft benötigen.

Weiters wirst du in weiterer Folge auch noch eine Möglichkeit zum Zusammensetzen einer Zeile benötigen:

```
string join(const vector<string>& data, char delimiter);
```

Implementiere diese gleich dazu.

4. Mit Hilfe dieser beiden Module kannst du jetzt daran gehen die eigentliche Funktionalität des Lesens einer CSV Datei zu implementieren. Erledige dies mittels eines neuen Moduls `csv_utilities`:

   ```
   vector<vector<string>>
   csv_reader(vector<string> lines, vector<string>& header);

   vector<string>
   csv_writer(vector<vector<string>> data, vector<string> header);
   ```

5. Jetzt fehlt noch die eigentliche Hauptfunktionalität, nämlich das Ermitteln des Erfolges. Dafür gehört ein Modul `calc_awards` her:

   ```
   vector<vector<string>> calc_awards(vector<vector<string>> data);
   ```

   Bedenke bei der Mittelwertberechnung, dass in C++ eine Division von ganzen Zahlen wieder eine ganz Zahl ergibt. `static_cast` ist hier eindeutig dein Freund!

   Ein String mit einer korrekten Zahl kann leicht mittels `stoi` in einen Integer gewandelt werden.

6. Wie schon angesprochen können hier jede Menge an Fehlern auftreten. Für jede Fehlerart ist nicht nur eine entsprechende Fehlermeldung auf `stderr` auszugeben, sondern auch ein jeweils unterschiedlicher Fehlercode (Statuscode) an den aufrufenden Prozess zurückzugeben!

   Wie bekannt bedeutet 0 in der Regel Erfolg. Damit bietet es sich an als andere Codes 1,2,3,... zu verwenden.

7. Unit-Tests wollen wir ausnahmsweise weglassen!

4 Übungszweck dieses Beispiels

- Übersetzen einer Fremdbibliothek (`fmt`) als statische Bibliothek mit `cmake` und `make`
- Verwenden und Linken einer statischen Bibliothek
- Lesen und Schreiben einer Textdatei bzw. Umgang mit Streams (`ifstream`, `ofstream`, `fstream`, `istringstream`, `ostringstream`, `tellg()`, `tellp()`, `seekg()`, `seekp()`, `good()`, `eof()`, `fail()`, `bad()`, `flush()`, `close()`)
- `const`
- `string` und splitten eines Strings
- einfache Umwandlung eines Strings in eine Zahl `stoi`
- `try`, `catch`, `runtime_error`, `throw_with_nested`, `what()`
- Exception-objekte per Referenz abfangen
- Exit-Codes je Fehlerart
- CSV-Dateien lesen und schreiben
- `static_cast` zur Typumwandlung einer ganzen Zahl in eine Gleitkommazahl für den Compiler