Beispiel `04_checksums`

Dr. Günter Kolousek

14. Juli 2016

1 Allgemeines

- Drucke dieses Dokument **nicht** aus!
- Halte **unbedingt** die Coding Conventions ein! Zu finden am edvossh!
- Lege für dieses Beispiel wieder ein entsprechendes Unterverzeichnis mit Beispielnummer und Beispielname an!
- Verwende den angegebenen Texteditor!

2 Aufgabenstellung

Schreibe ein `C++` Programm `checksums`, das für eine gegebene Datei verschiedene Checksummen berechnet. Je nach Eingabe sollen entweder diese Checksummen in einer eigenen Checksummendatei abgespeichert werden oder mit den Checksummen aus einer Checksummendatei verglichen werden.
Der Sinn dieses Programmes besteht darin Fehler (z.B. Übertragungsfehler) zu erkennen.
Die Bedienung soll folgendermaßen funktionieren:

```
$ checksums
Checksummen überprüfen (1)
Checksummen erstellen  (2)
Programm beenden       (q)
>>>                             <-- <Return> gedrückt!
>>>                             <-- <Return> gedrückt!
>>> 2

Dateiname: students.csv
Checksummen in students.csv.chk erstellt: 38544
```

```
Checksummen überprüfen (1)
Checksummen erstellen  (2)
Programm beenden       (q)
>>> 1

Dateiname: students.csv
Checksummen sind gleich!

Checksummen überprüfen (1)
Checksummen erstellen  (2)
Programm beenden       (q)
>>> a
Ungültige Eingabe!

Checksummen überprüfen (1)
Checksummen erstellen  (2)
Programm beenden       (q)
>>> q
$
```

3 Anleitung

1. Schreibe eine Funktion `uint8_t parity_bits_vertical(vector<uint8_t> data)` in einer Datei `checksums.cpp`, die für einen Vektor mit Elementen des Typs `uint8_t` ein Paritätsbyte ermittelt.

   Unter der Parität einer Zahl wird die Eigenschaft verstanden, ob diese gerade oder ungerade ist. Das Prinzip der Parität in Fehlererkennungskodierungen ist, dass ein zusätzliches Bit – nämlich das Paritätsbit –- an einen Datenblock von Bits hinzugefügt wird, das die Parität der Anzahl der Einsen im Datenblock angibt. Eine Eins zeigt an, dass es eine ungerade Anzahl von Einsen gibt und eine Null zeigt an, dass die Anzahl der Einsen gerade ist. Damit hat ein solcher Datenblock immer eine gerade Anzahl an Einsen.

   Bei einer Folge von Bytes (also genau genommen von Oktetts) betrachten wir jeweils alle Bits mit dem Index 7 als einen Datenblock, jeweils alle Bits mit dem Index 6 als einen Datenblock usw. D.h. wir betrachten jeweils die Spalten, wenn wir die Bytes untereinander schreiben.

   Das nachfolgende Beispiel zeigt für einen Datenblock aus vier Bytes in der letzten Zeile das zugehörige Paritätsbyte:

0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	0	0	1	0	1
1	0	1	0	0	0	1	0
1	0	0	0	0	1	1	1

Ein Tipp am Rande: Der xor-Operator `^` kann durchaus verwendet werden!

In `C++` ist die Größe eines `char` nicht spezifiziert (wie auch die Eigenschaft, ob `char` vorzeichenbehaftet ist oder nicht), obwohl man prinzipiell davon ausgehen kann, dass es für die gängigen Plattformen die Größe eines Bytes (also Oktetts) hat.

Allerdings wollen wir uns nicht darauf verlassen und ganz sicher sein, dass unser verwendeter Datentyp auch vorzeichenlos betrachtet wird. Da kommt `uint8_t` aus der Headerdatei `cstdint` ins Spiel, die mittels eines beherzten `#include <cstdint>` eingebunden wird. Wie man aus dem Namen leicht erahnen kann, handelt es sich um einen `unsigned int` mit der Größe von 8 Bits. Dieser Typ ist nur vorhanden, wenn das System solch einen Typ kennt, also in unserem Fall, eine vorzeichenlose Zahl bestehend aus insgesamt 8 Bits.

So jetzt hast du alle Informationen beisammen, um die Funktion zu implementieren. Los geht's!

Diese Funktion gehört natürlich auch getestet. Schreibe daher in der gleichen Datei auch eine `main` - Funktion, die die Funktion `parity_bits_vertical` aufruft und das Ergebnis ausgibt.

Hmm, vielleicht bemerkst du, dass das ausgegebene Ergebnis nicht so aussieht, wie du dir das vorgestellt hast? Tja, wandle den `uint8_t` mittels `static_cast<int>(x)` um.

Als Testdaten verwende

- `0x00`, `0xFF`
- `0x00`, `0x01`, `0x02`
- `0x00`, `0x01`, `0x02`, `0x03`
- `0x0F`, `0xF0`, `0x3C`

Stimmt das Ergebnis so wie du dir das ausgerechnet hast? Gut, dann weiter zum nächsten Punkt. Der Präfix `0x` in einem Zahlenliteral kennzeichnet eine hexadezimale Darstellung, aber das war ja leicht zu erraten, nicht wahr?

2. Allerdings ist die Ausgabe nicht *so* berauschend… Schöner wäre doch eine bitweise Ausgabe der jeweiligen Ergebnisse. Das kann folgendermaßen erreicht werden:

   ```
   cout << bitset<8>{0xC3} << end;
   ```

   Die Ausgabe dafür wäre:

```
11000011
```

Dafür ist die Headerdatei `bitset` einzubinden (wiederum mittels `#include <bitset>`). Was passiert hier? Es gibt anscheinend in der Standardbibliothek einen Datentyp `bitset`, der in der Headerdatei `bitset` deklariert wird. Diesen kann man mit einer ganzen Zahl initialisieren und bei der Ausgabe wird dieser bitweise ausgegeben. Natürlich kann dieser Datentyp mehr, aber das brauchen wir im Moment nicht.

3. Soweit ist das ja bis jetzt in Ordnung, aber wie sieht es mit anderen Checksummen aus? Die gehören auch noch implementiert. Sinnvollerweise werden die anderen in der gleichen Datei realisiert. Aber wie diese Funktionen verwendet werden, das ist vollständig unabhängig. Wir wollen diese in einem Menü aufrufbar machen und die Daten aus Dateien lesen und in Dateien schreiben, aber unter Umständen würde man diese Checksummen vielleicht auch gerne in einem Serverprogramm verwenden und die Daten vom Netzwerk lesen bzw. schreiben.

   D.h. es macht Sinn, das eigentliche Hauptprogramm in einer eigenen Datei `main.cpp` zu schreiben und die Implementierungen der Checksummen in der Datei `checksums.cpp` zu belassen.

   So, diese Änderung wollen wir in diesem Schritt jetzt vornehmen. Damit wird sich im Moment an der Funktionalität nichts ändern, aber wir restrukturieren unseren Sourcecode, sodass in weiterer Folge unsere Checksummenfunktionen in beliebigen Anwendungen verwendet werden können.

   Los geht's. Wenn du das jetzt erledigt hast, dann stellt sich in weiterer Folge die Frage wie zu übersetzen ist?

   Ganz einfach, so wie bisher, aber mit Angabe beider `.cpp` Dateien.

   Also, so soll es sein:

   ```
   $ g++ checksums.cpp main.cpp -o checksums
   ```

   Fertig? Hat nicht funktioniert? Welche Fehlermeldung hast du bekommen? Im nächsten Punkt geht es weiter.

4. Ich gehe davon aus, dass du es wirklich ausprobiert hast. Das ist ein Teil von deinem Lernerfolg.

   Du hast jede Menge Fehlermeldungen erhalten. Richtig? Wichtig ist, dass du dir die *erste* dieser Fehlermeldungen ansiehst. Die anderen sind oft einfach nur Folgefehler. Ja, das ist leider so.

   Der erste Fehler lautet in etwa so (abhängig von Compiler und eingestellter Sprache):

   ```
   main.cpp: In function ‘int main()’:
   main.cpp:10:70: error: ‘parity_bits_vertical’ was not declared in this scope
     cout << bitset<8>{parity_bits_vertical(vector<uint8_t>{0x00,0xFF})} << endl;
   ```

Diese Meldung sagt aus, dass in `main.cpp` in der Zeile 10 und der Spalte 70 ein Fehler beim Übersetzen aufgetreten ist, nämlich, dass `parity_bits_vertical` nicht deklariert ist. Das liegt daran, dass der Compiler beim Übersetzen von `main.cpp` die Funktion `parity_bits_vertical` einfach nicht kennt, da der Compiler jede Datei separat übersetzt.

D.h. wir müssen dem Compiler mitteilen wie diese Funktion aussieht.

Es wäre doch keine schlechte Idee einfach die Datei `checksum.cpp` in die Datei `main.cpp` einzubinden. Das kann ganz einfach mittels `#include "checksums.cpp"` realisiert werden. Füge diese Preprocessor-Anweisung gleich nach den anderen Include-Direktiven ein. Vielleicht ist dir aufgefallen, dass wir jetzt doppelte Anführungszeichen anstatt den spitzen Klammern verwendet haben. Die spitzen Klammern bedeuten, dass in den Systempfaden gesucht wird, während mit den Anführungszeichen in den Projektpfaden (also defaultmäßig im aktuellen Verzeichnis) nach der einzubindenden Datei gesucht wird.

Ok, wie sieht's aus? Hat alles funktioniert? Nein? Macht nichts, ich erkläre das im nächsten Punkt.

5. Du hast wieder eine Fehlermeldung erhalten, nicht wahr? Allerdings nur eine und die auch nicht vom Compiler sondern vom *Linker*. Die wird in etwa so aussehen:

```
/tmp/cc5VVY8v.o: In function `parity_bits_vertical(std::vector<unsigned char, std::al
main.cpp:(.text+0x0): multiple definition of `parity_bits_vertical(std::vector<unsign
/tmp/ccNV9Yxk.o:checksums.cpp:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

   Schau dir wieder die Fehlermeldung an. D.h. du musst diese *lesen*! Da steht so etwas von "`multiple definition of `parity_bits_vertical...`". Tja, das darf halt nicht sein. Eine Funktion darf nur genau ein Mal definiert sein.

   Hmm, aber wie kann ich dem Compiler bekanntgeben, dass er die Funktion kennt ohne, dass diese zwei Mal definiert ist? Hier kommt der elementare Unterschied zwischen Definition und Deklaration ins Spiel. D.h. wir benötigen eine Deklaration der Funktion `parity_bits_vertical` in der Datei `main.cpp`.

   Lösche die `#include "checksums.cpp"` Präprozessordirektive und füge nach deinem `using namespace std;` die folgende Deklaration hinzu: `uint8_t parity_bits_vertical(vector<uint8_t> data);`

   Jetzt's funktioniert es.

6. Das ist ja soweit in Ordnung, aber hat den entscheidenden Nachteil, dass eine Änderung in der `checksums.cpp` Datei auch eine manuelle Änderung in `main.cpp` nach sich zieht, auf die man nicht vergessen darf. Und mit furchtbaren Compilerfehlern bestraft wird…

Besser wäre es die Deklaration in eine Headerdatei `checksums.h` zu geben und diese Headerdatei sowohl in `checksums.cpp` als auch in `main.cpp` einzubinden. Dann kann der Compiler sowohl in `checksums.cpp` die Deklaration mit der Definition der Funktion `parity_bits_vertical` überprüfen als auch in `main.cpp` die Deklaration mit dem Aufruf.

Erstelle eine Datei `checksums.h` und verschiebe die Deklaration der Funktion `parity_bits_vertical` von `main.cpp` nach `checksums.h`. Füge weiters sowohl in `checksums.cpp` als auch in `main.cpp` jeweils die Include-Direktive `#include "checksums.h"` hinzu.

Denke daran, dass wir alle Include-Direktiven immer an den Anfang einer Datei stellen!

Wenn du jetzt übersetzt, wird es wieder Fehlermeldungen hageln. Probiere es einfach aus.

So, woran liegt es? Der Grund liegt daran, dass die Typen `vector` und `uint8_t` dem Compiler beim Übersetzen der Deklaration nicht bekannt sind, da wir keine `using namespace std;` Anweisung davor gesetzt haben. Das können wir natürlich leicht nachholen, aber das tun wir **nicht**! In eine Header-Datei geben wir keine `using namespace std;` oder ähnliches, da wir unseren Namensraum nicht *verschmutzen* wollen (wie du schon im Theorieunterricht gelernt hast).

Also ist es notwendig in der Headerdatei überall jeweils `std::` an die Typen vorne anzufügen. Dann wird es funktionieren.

Durch das Includieren von `checksums.h` in `checksums.cpp` als auch in `main.cpp` kann der Compiler einerseits überprüfen, ob die Verwendung von `parity_bits_vertical` mit der Deklaration übereinstimmt als auch ob die Deklaration mit der Definition übereinstimmt. Damit können Fehler schon vom Compiler gefunden werden.

7. Ok, ist schon ziemlich gut, aber es fehlt noch der Guard, der bei mehrfachen Einfügen der Headerdatei sicherstellt, dass der Inhalt real nur einmal eingebunden wird. Füge an den Anfang der Headerdatei:

   ```
   #ifndef CHECKSUMS_H
   #define CHECKSUMS_H
   ```

   An das Ende kommt:

   ```
   #endif
   ```

8. Lassen wir es vorerst mit dem Umorganisieren und wenden wir uns wieder dem eigentlichen Problem zu. Was ist das eigentliche Problem?

   Denken...

Ok, das eigentliche Problem ist, ein Programm zu schreiben, das Fehler in Daten erkennen kann.

Probieren wir es einfach aus. Erweitere dein Programm um das Berechnen der Checksumme für die Werte 0 und 255. Ok, das haben wir ja schon in der hexadezimalen Schreibweise getestet. In oktaler Schreibweise beginnt jede Zahl mit einer führenden 0. Also ist 0123 gleich $1 \cdot 8^2 + 2 \cdot 8 + 3$ und das ist in Dezimalschreibweise 83.

Für die Darstellung langer binärer Werte ist die hexadezimale Schreibweise besser geeignet als die oktale Schreibweise, aber für kurze wäre es doch bedeutend lesbarer den Wert auch wirklich binär anschreiben zu können. In `C++14` kann man jetzt den Präfix `0b` für Zahlen verwenden, also `0b11001010`. Was ergibt diese binäre Zahl in oktaler, in hexadezimaler und in dezimaler Schreibweise?

Gut, mit diesem Wissen kannst du jetzt einfach das Programm um die Berechnung der Checksumme für die Werte 0 und 255 erweitern. Nehmen wir weiters an, dass sich bei beiden Datenwerten jeweils das höchstwertigste Bit, auf Grund eines Übertragungsfehlers ändert. Programmiere auch diese Ausgabe hinzu. Was fällt dir auf?

9. Gut, wir gehen her und programmieren eine weitere Checksumme, die "Fletcher Checksumme" in der 16 Bit-Variante, aka "Fletcher-16".

   Die fertig ausprogrammierte Funktion sieht folgendermaßen aus:

```
uint16_t fletcher16(vector<uint8_t> data) {
    uint16_t sum1{};
    uint16_t sum2{};

    for (auto item : data) {
         sum1 = (sum1 + item) % 255;
         sum2 = (sum1 + sum2) % 255;
    }

    return (sum2 << 8) | sum1;
}
```

   Baue diese Funktion in `checksum.cpp` und `checksum.h` ein und rufe die Funktion mit einer geeigneten Ausgabe für die beiden problematischen Werte in `main` auf. Ok, diese Funktion erkennt ohne Probleme das einfache Vertauschen eines Bits (zumindest haben wir es an einem Beispiel sehen können).

10. Schaue dir einmal den Algorithmus etwas genauer an und arbeite den Algorithmus von Hand für `fletcher16(vector{0, 255})` und für `fletcher16(vector{255, 0})` ab! Was fällt dir auf?

Denken...

Fertig? Interessant nicht?

11. Es liegt aber nicht an der Reihenfolge wie du leicht bei der Berechnung von `fletcher16(vector{0xF0, 0x0F})` und `fletcher16(vector{0x0F, 0xF0})` sehen kannst. Es liegt nur an den 00 und FF Blöcken, die der Fletcher-16 Algorithmus nicht unterscheiden kann.

12. Bleiben wir noch bei der Implementierung dieses Algorithmus?

    - Warum haben die Variablen `sum1` und `sum2` den Datentyp `uint16_6`?
    - Was bewirkt die Modulooperation `% 255`?
    - Was wird bei dem Ausdruck `sum2 << 8 | sum1` erreicht?

13. Den "Logik-Teil" dieses Beispieles haben wir jetzt erledigt. Es fehlt noch der Ein-/Ausgabeteil. Dieser gliedert sich in eine Schnittstelle zum Benutzer und eine Schnittstelle zum Abspeichern in Dateien bzw. Lesen aus Dateien.

    Den derzeitigen Inhalt von `main` kannst du jetzt auskommentieren (oder auch löschen), dieser wird nicht mehr benötigt.

    Programmiere jetzt die Schnittstelle zum Benutzer gemäß den Vorgaben in der Aufgabenstellung. Halte dich genau an den Benutzerablauf, inklusive den Zeilenumbrüchen, dem Prompt und auch dem Verhalten, wenn nur die `<Return>`-Taste gedrückt wird.

    Damit nach dem Drücken der `<Return>`-Taste gleich wieder der Prompt erscheint, kannst du nicht mehr folgende Art von Code verwenden:

    ```
    char answer{};
    cin >> answer;
    ```

    Da ja der Eingabeoperator `>>` bis zum nächsten Nicht-Whitespace-Zeichen liest, funktioniert dies so nicht. Probiere es einfach aus und du weißt was gemeint ist.

    Die Lösung liegt darin, dass du eine ganze Zeile liest und dann den Inhalt dieser Zeile ansiehst:

    ```
    string answer;
    getline(cin, answer);
    ```

    Wenn du die `<Return>`-Taste drückst, dann enthält `answer` nur den Leerstring.

14. Als nächstes gehen wir die Ein- bzw. Ausgabe auf Dateiebene an. Diese Operationen sind prädestiniert dafür in eigenen Funktionen implementiert zu werden. Eigentlich kann man noch weiter gehen und diese sogar in einem eigenen Modul (also einer eigenen `.cpp`-Datei mit zugehöriger `.h` Datei) realisieren.

Gehe daher her und erstelle in einem Modul `file_utility` (`file_utility.cpp`, `file_utility.h` mit Guard) die Operationen:

```
vector<string> read_textfile(string filename);  // throws runtime_error
void write_textfile(string filename, vector<string> data)  // throws runtime_error
```

`vector<string>` enthält jeweils die Daten als Textzeilen.

Die Funktionen sollen einen `std::runtime_error` mit einer geeigneten Fehlermeldung werfen, wenn das Anlegen, das Auslesen oder das Beschreiben einer Textdatei nicht möglich ist (z.B. Rechte nicht vorhanden).

Teste dieser Funktionen!

Tipps zur Implementierung:

- Verwende `ifstream` aus der System-Include-Datei `fstream` folgendermaßen: `ifstream infile{"test.txt"}`
- Wenn die Datei nicht geöffnet werden konnte, dann hat die Bedingung `!infile` den Wert `true`.
- Die Funktion `getline(infile, line)` haben wir schon im Zusammenhang mit `cin` kennengelernt und leistet auch bei Dateien gute Dienste. Diese Funktion speichert die nächste Zeile in die Stringvariable `line` und liefert `infile` (als Referenz) zurück. Kann nicht gelesen werden, dann wird `infile` in den Fehlerzustand versetzt.
- `infile.close()` schließt die Datei.
- Verwende `ofstream` für die Ausgabe in eine Datei. Zur Ausgabe verwende ganz normal den `<<` Operator, aber vergiss nicht auf den Zeilenumbruch.
- Teste, dass auch die Exception geworfen wird: Recht zum Verändern des Verzeichnisses temporär wegnehmen bzw. Recht zum Verändern der Textdatei temporär wegnehmen.

15. So, jetzt wollen wir die Checksummen berechnen… Aber da tut sich ein kleines Problem auf: Unsere Checksummenfunktionen verarbeiten Typen von `uint8_t` aber die Funktionen zum Einlesen bzw. zum Schreiben verwenden Strings!

    D.h. es besteht einerseits die Möglichkeit die Checksumme als String in der Checksummendatei zu speichern und eine Konvertierung zwischen ganzen Zahlen und Strings vorzunehmen oder alle Daten binär in den Dateien abzulegen.

    Im Moment spricht alles dafür, die erste Variante auszuwählen, da wir die Dateibehandlung auf Basis von Textdateien schon fertiggestellt haben und andererseits die Konvertierung von Strings zu Zahlen schon können (die andere Richtung ist extrem einfach). Abgesehen davon können wir binäre Textdateien derzeit noch nicht lesen und auch nicht schreiben.

    Um die Implementierung davon vornehmen zu können, fehlt lediglich die Information wie man eine Zahl in einen String umwandeln kann. Hier kommt die Funktion

`string to_string(int)`, die einen Integer seine Stringdarstellung wandelt. Es gibt auch Überladungen dieser Funktion für andere arithmetischen Typen.

Damit müsste das Programm jetzt problemlos funktionieren!

4 Übungszweck dieses Beispiels

- `char` vs. `uint8_t`, `uint16_t`
- `0x`, `0b`, `0...`
- Bitoperationen `^`, `^=`, `|`, `<<`
- `<bitset>` kennenlernen
- Aufteilen in Source- und Headerdateien
- Compiler vs. Linker
- Deklaration vs. Definition
- Vorgegebenen Algorithmus analysieren
- Programmierung eines Textmenüs
- Textdateien öffnen, lesen, schreiben
- Rechte unter Linux einsetzen
- Verwendung eines Texteditors