Programmierkurs
Dr. Steffen Müthing
IWR, Universität Heidelberg

Aufgabenblatt 6
Abgabe 03. Dezember 2018
ipk-exercises:2018-ws-27-g5973cb3

Aufgabenblatt 6

Allgemeine Hinweise:

- Für die Aufgaben auf diesem Übungsblatt müssen Sie am 03.12. votieren.
- Dieses Blatt enthält viel Text, dies sind allerdings teils ausführliche Erklärungen – Sie müssen nicht sehr viel programmieren.
- Lesen Sie sich die Aufgaben jeweils vollständig durch, bevor Sie anfangen, sie zu bearbeiten!

Aufgabe 1

Statistiken

In dieser Aufgabe berechnen wir ein paar grundlegende Statistiken zu einer Menge von Zahlen.

Um Vektoren zu erstellen, mit denen Sie die Statistikfunktionen testen können, laden Sie folgende Dateien herunter:

https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2018/io.hh
https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2018/io.cc

Diese Dateien enthalten Routinen, um einen Vektor entweder von der Standardeingabe einzulesen oder zufällig zu erzeugen (siehe Kommentare in `io.hh`).

Aufgaben:

(a) Erstellen Sie drei Programme `readvector`, `uniform` und `normal`, die jeweils eine der drei unterschiedlichen Generatorfunktionen in `io.hh` verwenden und den Vektor dann auf die Standardausgabe schreiben.

Alle Funktionen in den folgenden Aufgaben sollen in wiederverwendbarer Weise geschrieben werden: Erstellen Sie einen Header `statistics.hh` und eine Implementierungsdatei `statistics.cc` für die Funktionen. Den Header includen Sie dann in jedem der drei Programme aus der ersten Aufgabe und wenden die Funktion(en) in jedem der drei Programme jeweils auf den eingelesenen Vektor an. Hierfür entfernen Sie am besten wieder den Code, der alle Vektoreinträge schreibt. Die Programme sollen immer alle schon implementierten Statistiken ausgeben.

(b) Erstellen Sie eine Funktion `double mean(const std::vector<double>& v)`, die den Mittelwert $\mathbb{E}[v]$ aller Einträge in dem Vektor v zurückliefert:

$$\mathbb{E}[v] = \mu = \frac{1}{N} \sum_{i=1}^{N} v_i,$$

wobei N die Anzahl der Einträge im Vektor angibt.

(a) Erstellen Sie eine Funktion `double median(const std::vector<double>& v)`, die den Median $M(v)$ aller Einträge in dem Vektor v zurückliefert. Um den Median zu berechnen, erstellen Sie eine sortierte Kopie $\tilde{v}$ von v und bestimmen $M(v)$ als

$$M(v) = \begin{cases} \tilde{v}_{\frac{N+1}{2}} & N \text{ ungerade} \\ \frac{1}{2}\left(\tilde{v}_{\frac{N}{2}} + \tilde{v}_{\frac{N}{2}+1}\right) & N \text{ gerade} \end{cases}$$

wobei N die Anzahl der Einträge im Vektor angibt. Beachten Sie die unterschiedlichen Indizierungsstrategien (1-basiert in der Formel oben, 0-basiert in C++) sowie den Spezialfall eines leeren Vektors!

(b) Erstellen Sie eine Funktion `double moment(const std::vector<double>& v, int k)`, die das k-te statistische Moment m_k aller Einträge in dem Vektor zurückliefert:

$$m_k = \mathbb{E}[v^k] = \frac{1}{N}\sum_{i=1}^{N} v_i^k,$$

wobei N die Anzahl der Einträge im Vektor angibt.

(c) Erstellen Sie eine Funktion `double standard_deviation(const std::vector<double>& v)`, die die Standardabweichung des Vektors berechnet:

$$s = \left(\mathbb{E}[(v-\mu)^2]\right)^{\frac{1}{2}}.$$

Überprüfen Sie, ob folgende Relation gilt:

$$\mathbb{E}[(v-\mu)^2] = \mathbb{E}[v^2] - \mathbb{E}[v]^2.$$

Aufgabe 2

CMake

In dieser Aufgabe erweitern Sie die vorherige Aufgabe um ein CMake-basiertes Buildsystem. Verwenden Sie hierfür die Informationen aus der Vorlesung.

Hinweis: Sie müssen nur die Teilaufgabe a) der vorherigen Aufgabe lösen. Falls Sie die anderen Teilaufgaben nicht lösen konnten, schreiben Sie einfach Dummy-Implementierungen für die diversen Statistik-Funktionen, die immer 0 zurückgeben.

Wichtig: Falls Sie die Aufgabe auf einem der Pool-Rechner bearbeiten, müssen Sie CMake mitteilen, dass Sie einen speziellen Compiler verwenden wollen:

```
cmake -DCMAKE_CXX_COMPILER=ipkc++ <weitere Optionen> <Pfad zu CMakeLists.txt>
```

Aufgaben:

(a) Erstellen Sie die Datei `CMakeLists.txt` wie in der Vorlesung beschrieben und fügen die drei Programme hinzu. Führen Sie `cmake` und `make` in einem Unterverzeichnis aus, um die Programme zu bauen.

(b) Verändern Sie verschiedene Header und Implementierungsdateien und führen Sie `make` erneut aus, um zu sehen, welche Dateien neu kompiliert und gelinkt werden. Um eine Datei als verändert zu markieren, können Sie einfach Leerzeilen hinzufügen / entfernen oder `touch <dateiname>` aufrufen.

(c) Wie Sie sehen, werden `io.cc` und `statistics.cc` für jedes Programm einzeln übersetzt. Legen Sie eine Bibliothek mit diesen beiden Dateien an, um die Mehrfachkompilierung zu vermeiden.

(d) Erstellen Sie ein Unterverzeichnis `release-build` und rufen dort ebenfalls CMake auf. Setzen Sie beim CMake-Aufruf den Build Type auf `Release` (siehe Vorlesung).

Vergleichen Sie die Laufzeit Ihrer Programme in den beiden Verzeichnissen für große, zufällig erzeuge Vektoren ($\approx 10^6 - 10^8$ Einträge).

(e) Erstellen Sie Tests für die Funktionen `mean()` und `median()`, die jeweils folgende Inputs testen:

- Einen leeren Vektor
- Einen im Test vorgegebenen Vektor mit 4 Einträgen
- Einen im Test vorgegebenen Vektor mit 5 Einträgen

Die Tests sollen jeweils überprüfen, ob die Funktion das korrekte Ergebnis zurückgibt und das wie in der Vorlesung beschrieben an CMake signalisieren. Wenn Sie möchten, können Sie dafür wie in der Vorlesung gezeigt die Funktion `assert()`[1] verwenden. Sie können entweder separate Programme für jeden Test erstellen oder nur jeweils ein Programm für jede Funktion. Binden Sie die Tests wie in der Vorlesung gezeigt ins Buildsystem ein und führen Sie sie mit `ctest` aus.

Hinweis: Falls Ihr Test wider Erwarten fehlschlägt, kann es sein, dass sich das Ergebnis aufgrund von Rundungsfehlern minimal unterscheidet. Vergleichen Sie zwei `double`-Werte am besten so:

```
#include <cmath>
double test_val = ...;
double reference_val = ...;
if (std::abs(test_val - reference_val) < 1e-10) {
  // test passed
}
```

Aufgabe 3

Häufigkeit von Buchstaben

Bisher haben Sie in den Übungen nur die Container `std::array` und `std::vector` verwendet, um Daten zu speichern. Diese beiden Datentypen modellieren eine Liste fixer Länge, bei der jeder Eintrag über einen 0-basierten, konsekutiven Index addressiert wird.

In vielen Fällen sind diese Datentypen völlig ausreichend, manchmal ist es jedoch praktisch, andere Werte als Zahlen (oder nicht-konsekutive Zahlen) zu verwenden, um auf Einträge zuzugreifen.

In dieser Aufgabe verwenden wir stattdessen Maps, um die Häufigkeit von Buchstaben bzw. Wörtern in einem Text auszuwerten.

Aufgaben:

(a) Schreiben Sie eine Funktion

```
std::map<char,int> get_frequencies();
```

die Buchstaben (Typ `char`) von der Standardeingabe liest, bis die Standardeingabe geschlossen wird, und zählt, wie oft die einzelnen Buchstaben vorkommen. Das Ergebnis soll sie im Rückgabewert speichern.

Um alle Buchstaben von der Standardeingabe einzulesen, verwenden Sie folgenden Codeschnipsel:

```
while (true)
{
  unsigned char c;
  // read in character
  std::cin >> c;
  // abort if input closed
  if (not std::cin)
    break;
  // work with c
  // PUT YOUR CODE THAT PROCESSES c HERE
}
```

(b) Schreiben Sie eine Funktion

```
void print_frequencies(const std::map<char,int>& frequencies);
```

die eine Liste von Buchstaben und zugehörigen Häufigkeiten auf die Standardausgabe ausgibt.

(c) Schreiben Sie ein Programm `letterfrequencies`, das die beiden Funktionen aus (a) und (b) benutzt, um die Buchstabenhäufigkeit eines Textes zu untersuchen. Hierzu soll es die beiden

[1] https://en.cppreference.com/w/cpp/error/assert

Funktionen einfach nacheinander aufrufen und den Rückgabewert der ersten Funktion als Parameter an die zweite Funktion weiterreichen.

Sie können Ihr Programm entweder testen, indem Sie Text in die Konsole tippen und die Eingabe mit `CTRL+D` beenden, oder Sie lassen Ihr Programm den Inhalt einer Datei verarbeiten:

```
./letterfrequencies < dateiname
```

(d) Wir sind eigentlich nur an Buchstaben interessiert, aber im Moment gibt Ihr Programm auch die Häufigkeit von Ziffern und Sonderzeichen aus. Verwenden Sie die Funktion[2]

```
bool std::isalpha(char c)
```

die `true` zurückgibt, wenn `c` ein Buchstabe ist, und überspringen Sie mit ihrer Hilfe alle Zeichen, die kein Buchstabe sind.

(e) Ihr Programm zählt Großbuchstaben getrennt von Kleinbuchstaben. Beheben Sie dies, indem Sie die Buchstaben mit der Funktion[3]

```
char std::toupper(char c)
```

in Großbuchstaben umwandeln und so Groß- und Kleinbuchstaben zusammen zählen. Sie können die Funktion auch mit einem Großbuchstaben aufrufen, dieser wird unverändert zurückgegeben.

(f) Ergänzen Sie Ihr Programm so, dass es abschliessend noch die Gesamtanzahl an **mitgezählten** Buchstaben ausgibt.

(g) Um die Ergebnisse unterschiedlich langer Texte vergleichbar zu machen, verändern Sie die Ausgabe so, dass statt der Anzahl der Anteil an allen Buchstaben ausgegeben wird, d.h. für den Buchstaben b geben Sie

$$p_b = \frac{f[b]}{\sum_{b' \in B} f[b']}$$

aus, wobei B die Menge aller gefundenen Buchstaben ist und f die Map mit den Häufigkeiten. **Wichtig:** Vor der Division müssen Sie eine der beiden Zahlen in `double` umwandeln, sonst bekommen Sie immer 0 als Ergebnis, weil der Compiler eine Ganzzahl-Division durchführt. Um einen Wert in `double` umzuwandeln, verwenden Sie folgende Syntax:

```
int b = 3;
int sum_all_b = ...;
p_b = static_cast<double>(b) / sum_all_b;
```

Hinweise:

- Um die Aufgabe nicht zu schwierig zu machen, werden Umlaute nicht korrekt verarbeitet. Verwenden Sie deshalb am besten englische Texte.
- Auf der Website https://gutenberg.org können Sie viele ältere Bücher als Text-Datei herunterladen, z.B.
  - die King-James-Bibel: https://www.gutenberg.org/files/30/30.txt,
  - Krieg und Frieden: https://www.gutenberg.org/files/2600/2600-0.txt,
  - Grimms Märchen: https://www.gutenberg.org/files/2591/2591-0.txt,
  - Moby Dick: https://www.gutenberg.org/files/2701/2701-0.txt,
  - Sherlock Holmes: https://www.gutenberg.org/files/1661/1661-8.txt.
- Bei großen Texten kann es sich lohnen, das Programm vom Compiler optimieren zu lassen, um die Laufzeit zu verbessern.
  - Wenn Sie Ihr Programm von Hand an der Kommandozeile übersetzen: Fügen Sie beim Kompilieren die Option `"-O3"` hinzu.

[2] http://en.cppreference.com/w/cpp/string/byte/isalpha
[3] http://en.cppreference.com/w/cpp/string/byte/toupper

– Wenn Sie CMake verwenden: Löschen Sie das Build-Verzeichnis und rufen Sie CMake erneut auf. Geben Sie CMake dabei die Option `"-DCMAKE_BUILD_TYPE=Release"` mit.

Aufgabe 4

Häufigkeit von Wörtern

Basierend auf Ihren Erfahrungen in der letzten Aufgabe sollen Sie jetzt statt der Häufigkeit von Buchstaben die von Wörtern zählen.

(a) Schreiben Sie eine Funktion

```
1 std::map<std::string,int> get_frequencies();
```

die Wörter (Typ `std::string`) von der Standardeingabe liest, bis die Standardeingabe geschlossen wird, und zählt, wie oft die einzelnen Buchstaben vorkommen. Hierzu können Sie sich am Gerüst der vorherigen Aufgabe orientieren und `char` durch `std::string` ersetzen.

Um Sonderzeichen und ähnliches zu entfernen, filtern Sie die Wörter mit Hilfe der Funktion `sanitize_word()` von der Vorlesungs-Homepage[4] [5]. Testen Sie nach dem Filtern, ob der resultierende String leer ist (`size()` == 0). Leere Strings sollen nicht mitgezählt werden!

(b) Schreiben Sie eine Funktion

```
1 void print_frequencies(const std::map<std::string,int>& frequencies);
```

die eine Liste von Wörtern und zugehörigen relativen Häufigkeiten auf die Standardausgabe ausgibt.

(c) Schreiben Sie ein Programm `wordfrequencies`, das die beiden Funktionen benutzt, um die Worthäufigkeit eines Textes zu untersuchen. Hierzu soll es die beiden Funktionen einfach nacheinander aufrufen und den Rückgabewert der ersten Funktion als Parameter an die zweite Funktion weiterreichen.

(d) Ersetzen Sie die `std::map` durch `std::unordered_map` und testen Sie, ob das Programm bei sehr großen Texten schneller wird.

[4] https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2018/sanitizeword.hh
[5] https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2018/sanitizeword.cc