Dr. U. Köthe *Heidelberg, 23. November 2015*

Übungsblatt 5

Anmerkung: Bitte achten Sie von diesem Zettel an darauf, dass Ihre Programme übersichtlich formatiert sind. Dies bedeutet unter anderem:

- Verwenden Sie selbsterklärende Variablennamen! Variablennamen aus nur einem Buchstaben sind nur für Schleifenzähler und einfache arithmetische Ausdrücke akzeptabel.
- Deklarieren und initialisieren Sie Variablen erst dort, wo sie verwendet werden! Andernfalls können Bugs entstehen, weil man inzwischen vergessen hat, welchen Wert eine Variable enthält oder in welcher Umgebung sie definiert war. (Dies war ursprünglich eine Erweiterung von C++ gegenüber C, wo Variablen immer am Anfang eines Blocks definiert werden mussten. Ab der Standard-Version 1999 hat C die Idee übernommen.)
- Rücken Sie Blöcke einheitlich ein! Viele häßliche Bugs werden von vornherein vermieden, weil sie im gut formatierten Code sofort auffallen. Eine Erklärung üblicher Einrückungsstile gibt es z.B. hier:
  `http://www.terminally-incoherent.com/blog/2009/04/10/the-only-correct-indent-style/`
  Ich bevorzuge, wie die Mehrheit der Leser dieser Seite, den Allman-Stil. Der K&R-Stil ist ebenfalls akzeptabel, die übrigen Vorschläge haben sich nicht durchgesetzt. Machen Sie einen Stil zur automatisierten Gewohnheit! Wenn nichts mehr hilft, unterstützt Ihr Editor wahrscheinlich eine Tastenkombination "Prettify" oder ähnlich, die den Code automatisch formatiert.
- Verwenden Sie zur Einrückung nur Leerzeichen, niemals Tab(ulator)-Zeichen! Die Einrückungstiefe von Tabulatoren ist nicht einheitlich eingestellt, so dass Ihr Code in einem anderen Editor oder bei einem Leser mit anderer Tabulatoreinrückung wie Kraut und Rüben aussieht. Ihr Editor hat wahrscheinlich eine Option "Tabulator durch x Leerzeichen ersetzen" oder ähnlich. Aktivieren Sie diese Option mit einer passenden Wahl für x (typisch: $x = 4$). Dann fügt der Editor automatisch Leerzeichen ein, wenn Sie die Tab-Taste drücken.
- Kommentieren Sie Ihr Programm!

Aufgabe 5.1 Zahlen in Array einsortieren [8 Punkte]

Schreiben Sie ein Programm `ordered_insert.cpp`, das natürliche Zahlen in ein Array vom Typ `std::vector<int>` einsortiert. Das Programm soll eine Schleife enthalten, die in jeder Iteration eine Zahl von der Konsole einliest und diese an der richtigen Position in das Array einfügt, so dass das Array stets aufsteigend sortiert bleibt. Wurde eine negative Zahl eingegeben, soll die Schleife sofort beendet und das Array zur Kontrolle ausgegeben werden. Sie dürfen in dieser Aufgabe *nicht* `std::sort()` verwenden! Zur Erinnerung: eine ganze Zahl wird durch folgende Kommandos von der Konsole gelesen:

```
int z;
std::cin >> z;
```

Aufgabe 5.2 Perfect Shuffle [16 Punkte]

Beim Mischen eines Kartenblatts gibt es unter versierten Pokerspielern und Zauberkünstlern die Technik des sogenannten "Perfect Shuffle". Dabei wird das Kartenblatt (bestehend aus 52 Karten) in zwei exakt gleich große Teile geteilt und so gemischt, dass immer abwechselnd eine Karte von jedem dieser Stapel genommen wird. Dabei kann man zwei Vorgehensweisen unterscheiden:

- Perfect-Out-Shuffle: $ABCDEFGH \Rightarrow AEBFCGDH$ (oberste Karte bleibt oben)
- Perfect-In-Shuffle: $ABCDEFGH \Rightarrow EAFBGCHD$ (mittlere Karte nach oben)

Der Witz der Methode ist es, dass die Karten nur scheinbar gemischt werden. Wiederholt man nämlich das "Perfect Shuffle" (fehlerfrei!) einige Male, kehrt der Stapel in den Ausgangszustand zurück – die perfekte Tarnung für Tricks und Betrügereien. Ihre Aufgabe ist es, ein Programm zu schreiben, das diese beiden Mischmethoden simuliert, um die Frage zu beantworten: Wie oft muss man das Kartenblatt bei ausschließlicher Verwendung von Perfect-In bzw. Perfect-Out mischen, um wieder in den Ausgangszustand zu gelangen?

Als Datenstruktur verwenden wir ein Array vom Typ `std::vector<int>`, das den Stapel durch die Zahlen $0...51$ repräsentiert.

(a) Implementieren Sie eine Funktion `std::vector<int> init_deck()`, die ein Array erzeugt, das die aufsteigenden Zahlenfolge $0...51$ (als Repräsentation des Ausgangszustands) enthält.

(b) Implementieren Sie eine Funktion `bool check_deck(std::vector<int> cards)`, die genau dann `true` zurückgibt, wenn das übergebene Array im Ausgangszustand ist. Testen Sie mit `assert()`, dass die Funktion unmittelbar nach `init_deck()` den Wert `true` liefert.

(c) Implementieren Sie eine Funktion
`std::vector<int> shuffle(std::vector<int> cards, bool out)`,
die einen Durchgang des Mischens ausführt und das gemischte Array zurückgibt. Ist der Parameter `out=true`, soll ein Out-Shuffle ausgeführt werden, andernfalls ein In-Shuffle.

(d) Implementieren Sie je eine Schleife für In- und Out-Shuffle, die solange `shuffle()` ausführt, bis der Stapel wieder im Ausgangszustand ist. Geben Sie die benötigte Anzahl an Iterationen aus, die in jedem der beiden Fälle benötigt wurde.

Geben Sie Ihre Lösung im File `perfect_shuffle.cpp` ab.

Aufgabe 5.3 Wörter zerwürfeln [16 Punkte]

In dieser Aufgabe soll der Umgang mit Schleifen und den Datentypen `std::string` und `std::vector` geübt werden. Dazu wollen wir ein psychologisches Experiment überprüfen: "Nach einer Studie einer englischen Universität ist es egal, in welcher Reihenfolge die Buchstaben in einem Wort stehen. Das einzig Wichtige dabei ist, dass der erste und

letzte Buchstabe am richtigen Platz sind." (Quelle: Graham Ernest Rawlinson: The significance of letter position in word recognition. PhD Thesis, Psychology Department, University of Nottingham, Nottingham 1976, siehe `https://de.wikipedia.org/wiki/Buchstabensalat`). Die Lösung soll in der Datei `mix.cpp` abgegeben werden.

(a) Implementieren Sie eine Funktion `std::vector<std::string> split_words(std::string s)`, die den übergebenen String bei Leerzeichen trennt und ein Array mit den einzelnen Wörtern zurückgibt. Dabei werden Satzzeichen so behandelt, als würden sie zu dem Wort gehören, bei dem sie stehen.

Beispiel: Beim Aufruf von

```
split_words("Alpha beta! Gamma")
```

soll das Array mit den Elementen

```
{"Alpha", "beta!", "Gamma"}
```

zurückgegeben werden.

(b) Schreiben Sie eine Funktion `std::string mix(std::string s)`. Diese Funktion soll zunächst den ersten und den letzten Buchstaben (`a-z`, `A-Z`) in dem übergebenen String finden (eventuelle Satzzeichen müssen dabei übersprungen werden, denn sie sind keine Buchstaben). Anschließend sollen alle Zeichen, die zwischen diesen beiden Buchstaben stehen, mit Hilfe von `std::random_shuffle()` (aus dem Header `<algorithm>`) zufällig durchgemischt werden. Die beiden gefundenen Buchstaben sollen ihre Position jedoch behalten. *Beispiele:*

- `mix("alpha")` ⇒ `"ahlpa"`
- `mix("beta!")` ⇒ `"btea!"`
- `mix("informatik...")` ⇒ `"iomnartifk..."`

Da die neue Reihenfolge zufällig ist, bekommen Sie natürlich bei jedem Aufruf von `mix()` ein anderes Ergebnis.

(c) Erstellen Sie die Funktion `std::string split_and_mix(std::string s)`, die den übergebenen String zunächst in einzelne Wörter zerlegt (`split_words()`) und anschließend die Buchstaben in jedem Wort durcheinander mischt (`mix()`). Danach soll der String wieder zusammengesetzt werden, indem die Wörter mit Leerzeichen getrennt aneinander gehängt werden. Der zusammengesetzte String wird schließlich zurückgegeben.

(d) In Moodle finden Sie die Datei `text.hpp`. Kopieren Sie diese Datei in dasselbe Verzeichnis wie `mix.cpp` und inkludieren Sie sie mittels `#include "text.hpp"`. Sie können nun die Variablen `str1` bis `str5` in ihrem Code verwenden. Wenden Sie `split_and_mix()` auf diese fünf Strings an und geben Sie

```
urprünglicher Text in str1

Ergebis von split_and_mix(str1)

urprünglicher Text in str2
```

```
Ergebis von split_and_mix(str2)

... usw.
```

aus.

Bitte laden Sie Ihre Lösung spätestens bis 30. November 2016, 9:00 Uhr in Moodle hoch.