Dr. U Köthe *Heidelberg, 15. November 2016*

Übungsblatt 4

Anmerkung: Beginnend mit diesem Blatt dürfen Sie die Techniken der *prozeduralen Programmierung* verwenden, insbesondere Schleifen und die `if`-Anweisung.

Aufgabe 4.1 Kubikwurzel [9 Punkte]

In der Vorlesung haben wir behandelt, wie man mit dem Newtonverfahren die Quadratwurzel einer Zahl y berechnen kann. Am Ende der Aufgabe wird dies noch einmal kurz zusammengefasst. Hier sollen Sie das Gelernte auf die Kubikwurzel übertragen.

(a) Leiten Sie die Iterationsvorschrift und die Abbruchbedingung für die Berechnung der Kubikwurzel her.

(b) Implementieren Sie eine Funktion `double cbrt(double y)` entsprechend. Für negative Argumente soll die Funktion negative Werte zurückgeben. Überprüfen Sie mittels `assert()` für vier positive Zahlen `y`, dass die Ausgabe Ihrer Funktion mit `std::pow(y, 1.0/3.0)` übereinstimmt (möglicherweise erfordern diese Vergleiche eine kleine Toleranz, weil `std::pow()` und `cbrt()` unterschiedliche Algorithmen verwenden). Testen Sie außerdem, dass `cbrt(-y) == -cbrt(y)` gilt. Geben Sie die Lösung in einer Datei `kubikwurzel.cpp` ab.

Wiederholung: Newtonverfahren für die Quadratwurzel. Das Newtonverfahren dient dazu, mittels iterativer Verbesserung eine Nullstelle für eine gegebene Funktion $f(x)$ zu finden, d.h. man sucht ein x^*, so dass $f(x^*) = 0$ gilt. Kennt man eine Kandidatenlösung $x^{(t)}$, berechnet man eine neue Kandidatenlösung $x^{(t+1)}$ nach der Vorschrift

$$x^{(t+1)} = x^{(t)} - \frac{f\left(x^{(t)}\right)}{f'\left(x^{(t)}\right)}$$

wobei $f'(x)$ die erste Ableitung der gegebenen Funktion ist (wir setzen hier voraus, dass $f'(x) \neq 0$ für alle relevanten x gilt). Setzt man $x^{(t)} = x^*$ ein, folgt wegen $f(x^*) = 0$ auch $x^{(t+1)} = x^*$. Man sagt, dass x^* ein *Fixpunkt* der Iteration ist. Im Fall $x^{(t)} \neq x^*$ ist die neue Lösung $x^{(t+1)}$ tatsächlich eine Verbesserung – d.h. es gilt $\left|f\left(x^{(t+1)}\right)\right| < \left|f\left(x^{(t)}\right)\right|$ – wenn $x^{(t)}$ bereits "in der Nähe" von x^* liegt. Bei Quadrat- und Kubikwurzeln trifft das auf alle x zu, die dasselbe Vorzeichen wie x^* haben. Im Allgemeinen ist eine präzise Definition von "in der Nähe" (die Herleitung der sogenannten Konvergenzbedingungen) jedoch komplizierter.

Um die Quadratwurzel $x^* = \sqrt{y}$ zu bestimmen, setzt man $f(x) = x^2 - y$ und erhält

$$x^{(t+1)} = \frac{x^{(t)} + y/x^{(t)}}{2}$$

Da Computer nur eine endliche Rechengenauigkeit haben, wird man $f(x) = 0$ in der Regel nicht exakt erreichen. Deshalb bricht man die Iteration ab, sobald $|f(x)| \leq \epsilon$ mit einer kleinen Toleranz ϵ erfüllt ist (z.B. $\epsilon = 10^{-15}|y|$, wenn die Berechnungen mit dem Typ `double` durchgeführt werden).

Aufgabe 4.2 Pythagoreische Tripel [9 Punkte]

In einem rechtwinkligen Dreieck mit den Seitenlängen a, b und c gilt bekanntlich der Satz des Pythagoras

$$a^2 + b^2 = c^2$$

Sind alle Seitenlängen ganzzahlig, bezeichnet man die Zahlen (a, b, c) als *Pythagoreisches Tripel*. Das kleinste und bekannteste Pythagoreische Tripel ist $(3, 4, 5)$. Wir wollen in dieser Aufgabe einen Algorithmus erstellen, der weitere Tripel findet. Geben Sie die Lösung im File `pythagoras.cpp` ab.

(a) Schreiben Sie eine Funktion `bool is_square(int n)`, die testet, ob `n` eine Quadratzahl ist. Verwenden Sie dabei `std::sqrt()` zur Berechnung der (reellwertigen) Wurzel und `std::floor()` zum Abrunden auf die nächste ganze Zahl. Nutzen Sie aus, dass `std::floor()` keinen Effekt hat, wenn das Argument bereits eine ganze Zahl war.

(b) Optimieren Sie die Funktion `is_square()` aus (a), indem Sie ausnutzen, dass für alle Quadratzahlen gilt:

$$n \text{ ist Quadratzahl } \Rightarrow (n \bmod 4) \in \{0, 1\}$$

Alle Zahlen, die bei Division durch 4 einen anderen Rest liefern (also Rest 2 oder Rest 3), sind keinesfalls Quadratzahlen. Mit diesem Test kann `is_square()` für 50% der Eingaben sofort `false` zurückgeben, ohne das (relativ teure) `std::sqrt()` aufrufen zu müssen.

Beweisen Sie die Gültigkeit der obigen Aussage mit Hilfe der allgemeinen Eigenschaft der Modulo-Operation

$$(m \cdot n) \bmod k = ((m \bmod k) \cdot (n \bmod k)) \bmod k$$

und implementieren Sie die so optimierte Funktion `is_square()`.

Hinweis, der in ähnlicher Form für alle Übungsaufgaben und die Klausur gilt: Die optimierte Implementation können Sie natürlich auch abgeben, wenn Ihnen der Beweis nicht gelingt – dann bekommen Sie einfach die entsprechende Teilpunktzahl. Wenn Sie die ganze Teilaufgabe (b) nicht lösen können, arbeiten Sie in (c) einfach mit der `is_square()`-Originalversion aus (a) weiter.

(c) Schreiben Sie eine Prozedur `void pythagorean_triple(int bmax)`, die mit Hilfe von `is_square()` alle Pythagoreischen Tripel findet und auf die Konsole ausgibt, bei denen $a < b \leq$ `bmax` gilt. Rufen Sie die Prozedur in `main()` mit `bmax = 400` auf.

Tipp: Verwenden Sie eine sogenannte *zweifach geschachtelte* `for`–Schleife, d.h. eine äußere Schleife läuft über b, und im Körper dieser Schleife läuft eine zweite ("innere") Schleife über a.

Aufgabe 4.3 Palindrome [8 Punkte]

Palindrome sind Wörter oder Sätze, die von vorne wie von hinten gelesen gleich sind. Dabei werden Groß- und Kleinschreibung sowie Leer- und Satzzeichen ignoriert. Beispiele sind Otto, Anna, Rentner, Lagerregal und "Ein Neger mit Gazelle zagt im Regen nie". Sie sollen hier eine Funktion schreiben, die testet, ob ein gegebener String ein Palindrom ist. Geben Sie die Lösung im File `palindrom.cpp` ab.

(a) Implementieren Sie eine Funktion `std::string to_lower(std::string s)`, die einen neuen String zurückgibt, bei dem alle im String `s` vorkommenden Großbuchstaben in Kleinbuchstaben konvertiert wurden. Benutzen Sie dazu die Funktion `std::tolower()` (mit `#include <cctype>`), die ein einzelnes Zeichen umwandelt (informieren Sie sich in der C++ Referenz über Details). Prinzipiell kann diese Funktion auch Umlaute korrekt umwandeln, es hängt jedoch stark von Ihren Rechnereinstellungen und dem verwendeten Compiler ab, ob das auch verlässlich funktioniert.

(b) Schreiben Sie eine Funktion `std::string letters_only(std::string s)`, die aus dem String `s` alle Zeichen entfernt, die keine Buchstaben sind, und das Ergebnis zurückgibt. Benutzen Sie dafür die Funktion `std::isalpha()`.

(c) Implementieren Sie unter Verwendung der beiden anderen Funktionen nun die Funktion `bool is_palindrom(std::string s)`, die genau dann `true` zurückgibt, wenn `s` ein Palindrom ist. Testen Sie an einigen Beispielen und Gegenbeispielen die Korrektheit mittels `assert()`.

(d) Implementieren Sie außerdem die Möglichkeit, dass der Benutzer auf der Kommandozeile ein Wort oder einen Satz eingibt, und Ihr Programm mit "ist ein Palindrom" oder "ist kein Palindrom" antwortet. Tipp: Um einen String von der Kommandozeile einzulesen, können Sie folgenden Code verwenden:

```
std::cout << "Gib ein Wort oder einen Satz ein:\n";
std::string s;
std::getline(std::cin, s);
```

Aufgabe 4.4 E-Mail-Adressen [14 Punkte]

In dieser Aufgabe soll der Umgang mit dem String-Datentyp geübt werden. Dazu wird eine Funktion implementiert, die überprüft, ob ein String die Form einer E-Mail-Adresse hat. Dabei verwenden wir jedoch nicht den kompletten Regelsatz für E-Mail-Adressen (`http://de.wikipedia.org/wiki/E-Mail-Adresse`), sondern nur ein paar ausgewählte Regeln.

Implementieren Sie die Funktion `bool is_email(std::string s)`, die genau dann `true` zurückgibt, wenn der String `s` die vereinfachten Regeln für E-Mail-Adressen erfüllt:

Eine E-Mail-Adresse hat die Form `"*@*.*"`, wobei es sich bei `*` um eines oder mehrere "erlaubte" Zeichen handelt. Erlaubte Zeichen sind `a-z`, `A-Z`, `0-9`, sowie `._-` (Punkt, Unterstrich, Strich). Es müssen die folgenden Regeln erfüllt sein:

- Es kommt genau ein @ Zeichen vor.
- Rechts von @ steht mindestens ein Punkt.
- Links von @ steht mindestens ein erlaubtes Zeichen.
- Zwischen @ und dem letzten vorkommenden Punkt steht mindestens ein erlaubtes Zeichen.
- Rechts vom letzten Punkt steht mindestens ein erlaubtes Zeichen, das kein Punkt ist.

Tipp: Überprüfen Sie zunächst, ob in dem kompletten String nur erlaubte Zeichen und @ stehen. Literale für einzelne Zeichen werden in C++ als `'z'` geschrieben (also in einfachen Anführungszeichen, im Gegensatz zu doppelten Anführungszeichen bei Stringliteralen, also Zeichen*ketten*). Gehen Sie anschließend die obigen Regeln durch. Die folgenden

String-Funktionen können bei der Bearbeitung helfen (`s` ist eine Stringvariable, `zeichen` ein einzelnes Zeichen, `k`, `start` und `length` sind ganze Zahlen):

- `s.size()`: Gib die Länge des Strings `s` (d.h. die Anzahl der Zeichen) zurück.
- `s[k]`: Gib das Zeichen zurück, das sich in `s` an der Indexposition `k` befindet.
- `s.substr(start, length)`: Gib den Teilstring der Länge `length` aus `s` zurück, der beim Index `start` beginnt.
- `s.substr(start)`: Gib den Teilstring aus `s` zurück, der beim Index `start` beginnt und bis zum Ende von `s` geht.
- `s.find(zeichen)`: Suche von links nach rechts den Index, wo das `zeichen` erstmals vorkommt, und gib diesen Index zurück. Wenn `zeichen` nicht in `s` vorkommt, gib die spezielle Konstante `std::string::npos` (für "Index ungültig") zurück. Beispiel: Wenn `s` das Zeichen @ nicht enthält, liefert der Ausdruck

  ```
  s.find('@') == std::string::npos
  ```

  den Wert `true`, andernfalls `false`.
- `s.rfind(zeichen)`: Dasselbe, aber suche von rechts nach links.
- `std::isalnum(zeichen)`: Gib genau dann `true` zurück, wenn `zeichen` ein Buchstabe oder eine Ziffer ist. Diese Funktion befindet sich im Standardheader `<cctype>`.

Informationen über weitere Details finden Sie in der C++ Referenz.

Testen Sie Ihre Funktion mittels `assert()` für sechs gültige und sechs ungültige E-Mail-Adressen. Damit Ihre Tests aussagekräftig sind, müssen Sie verschiedene gültige Email-Formen (z.B. mit mehreren Punkten vor oder nach dem @) und unterschiedliche Fehlervarianten verwenden. Geben Sie Ihre Lösung im File `email.cpp` ab.

Bitte laden Sie Ihre Lösung spätestens bis 23. November 2016, 9:00 Uhr in Moodle hoch.