Dr. Ole Klein, Dr. Steffen Müthing
IWR, Universität Heidelberg

Abgabe 8. Dezember 2017

Allgemeine Hinweise:

- **Am Freitag, den 1.12. findet die Vorlesung statt!**

Übung 1 *Flächenberechnung für Polygone*

Ein Polygon ist eine geometrische Figur, die aus einem geschlossenen stückweise linearen Streckenzug besteht. Dazu werden n Punkte $p_i = (x_i, y_i), i \in \{0, \dots, n-1\}$ in der Ebene $\mathbb{R}^2$ definiert und jeweils eine gerade Linie von p_i zu p_{i+1} gezogen. Geschlossen wird die Kurve, indem man abschließend eine Linie von p_{n-1} zu $p_n := p_0$ zieht. Das zweidimensionale Volumen (d.h. die Fläche) eines solchen Polygons ist gegeben durch

$$A = \frac{1}{2} \cdot \sum_{i=0}^{n-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

mit dem letzen Punkt p_n wie oben definiert. Wenn die Punkte im mathematisch positiven Sinne angegeben sind, entspricht das der üblichen Vorstellung eines Flächenmaßes, bei mathematisch negativer Angabe ("Uhrzeigersinn") ist A negativ.

Schreiben Sie ein Programm, das das Volumen eines gegebenen Polygons berechnet. Gehen Sie dabei wie folgt vor:

1. Schreiben Sie ein Struct `Point`, das zwei **`double`** zum Speichern der Koordinaten eines Punktes verwendet.
2. Schreiben Sie ein Struct `Polygon`, das ein Tupel (also einen Vector) von `Points` enthält.
3. Schreiben Sie eine Funktion

```
Polygon construct(const std::vector<double>& x,
                  const std::vector<double>& y
                  )
```

   die die Einträge von `x` als x-Koordinaten und die von `y` als y-Koordinaten interpretiert, daraus `Points` baut, und mit diesen wiederum ein `Polygon` konstruiert. Achten Sie dabei darauf, dass Sie fehlerhafte Eingaben entsprechend abfangen!
4. Schreiben Sie eine Funktion **`double`** `volume(`**`const`** `Polygon& polygon)`, die für ein gegebenes Polygon `polygon` mit der obigen Formel das Volumen bestimmt. Stellen Sie sicher, dass diese Funktion auch für "Polygone" mit 0, 1 oder 2 Ecken funktioniert, wobei wir in diesen Fällen $A := 0$ setzen.

Testen Sie Ihre Implementierung, indem Sie die Volumina der ersten 10 regelmäßigen n-Ecke berechnen. Die i-te Koordinate eines regelmäßigen n-Ecks ist durch die folgende Formel gegeben:

$$p_i = \left(\cos\left(\frac{i}{n} \cdot 2\pi\right), \sin\left(\frac{i}{n} \cdot 2\pi\right) \right).$$

Die trigonometrischen Funktionen (`std::sin(`**`double`**`)` etc.) sind in der Standardbibliothek im Header `cmath` definiert, π müssen Sie als Konstante selbst definieren:

```
1 const double pi = M_PI;
```

Für solche regelmäßigen n-Ecke gilt

$$A = \frac{n}{2} \cdot \sin\left(\frac{2\pi}{n}\right),$$

und damit für eine Auswahl der ersten n-Ecke:

Ecken	1	2	3	4	6	8
Fläche	0	0	$\frac{3}{4} \cdot 3^{1/2}$	2	$\frac{3}{2} \cdot 3^{1/2}$	$2 \cdot 2^{1/2}$

(Grundlagen)

Übung 2 *Wohlgeformte Ausdrücke*

Um den von Ihnen geschriebenen Programmcode zu interpretieren, muss der Compiler aus einer langen Folge von einzelnen Zeichen Sinn konstruieren, und dabei z.B. erkennen, wo eine Funktion anfängt und wo sie aufhört, und wann Zeichenketten gleichen Inhalts die gleiche Variable bezeichnen (das hängt vom Kontext ab). Ähnliches passiert bei einem modernen Taschenrechner, der bei der Zeichenkette "$(3 + 4) \cdot (5 + 2)$" erkennen muss, in welcher Reihenfolge Addition und Multiplikation durchzuführen sind.

Ein einfacher Spezialfall der dabei auftretenden Problemstellungen ist das Überprüfen, ob die im Ausdruck auftauchenden Klammern wohlgeformt sind: jede sich öffnende Klammer braucht eine passende sich schließende Klammer, und umgekehrt. Während der obige Ausdruck in diesem Sinne wohlgeformt ist, fehlt z.B. bei "$1 + 7) \cdot 3$" eine der beiden Klammern. Schreiben Sie die folgenden Funktionen, die Ausdrücke auf korrekte Klammerung überprüfen. Dabei können Sie im Rahmen dieser Aufgabe alle weiteren auftretenden Zeichen ignorieren.

(a) Schreiben Sie eine Funktion `bool check_parentheses(std::string symbols)`, die für eine Zeichenkette namens `symbols` prüft, ob die auftretenden `'('` und `')'` konsistent sind, d.h. nie mehr Klammern geschlossen werden als offen sind, und am Ende der Zeichenkette alle Klammern geschlossen wurden. Wie können Sie das durch Mitzählen erreichen?

Auf der Homepage finden Sie einen Header `input.hh`[1] und eine Quellcodedatei `input.cc`[2]. Diese definieren eine Funktion `std::string read_stream(std::istream&)`, die einen beliebigen String von dem gegebenen Stream einliest (dieser kann auch aus mehreren Zeilen bestehen).

(b) Schreiben Sie ein Programm `checkparens`, das mit Ihrer Funktion aus dem ersten Teil die Standardeingabe auf gültige Klammern überprüft. Benutzen Sie zum Bauen ein Buildsystem mit CMake.

Hinweise:

- Die Funktion `check_parentheses()` soll in einen Header `lint.hh` und eine Datei `lint.cc`.
- Zum Überprüfen der Klammern können Sie entweder mehrere `if`-Statements oder ein `switch`-Statement[3] verwenden.
- Der Funktion `read_stream()` können Sie als Parameter einfach `std::cin` übergeben.
- Um Ihr Programm zu testen, können Sie entweder etwas an der Konsole eintippen und Ihre Eingabe mit der Tastenkombination `CTRL+D` (das bedeutet end-of-file (EOF)) beenden, oder Sie überprüfen den Inhalt einer Datei, indem Sie die sogenannte *Eingabeumleitung* verwenden:

```
1 ./checkparens < dateiname
```

[1] https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2017/input.hh
[2] https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2017/input.cc
[3] http://en.cppreference.com/w/cpp/language/switch

Übung 3 *Wohlgeformte Ausdrücke für Fortgeschrittene*

Erweitern Sie die Funktion `bool check_parens(std::string symbols)` aus der vorherigen Aufgabe, so dass sie das gleiche für die Klammernpaare `'('` und `')'`, `'['` und `']'`, und `'{'` und `'}'` macht. Dabei muss für jede sich öffnende Klammer auch eine sich schließende Klammer des gleichen Typs auftreten.

Hierbei müssen alle in einem Klammernpaar stehenden Klammern vollständig aufgelöst sein, "([)]" ist also im Gegensatz zu "({}[{}])" kein wohlgeformter Ausdruck, weil das Zeichen `']'` außerhalb der beiden runden Klammern steht. Um diese Bedingung einhalten zu können, müssen Sie über alle bisher aufgetretenen Klammerpaare Buch führen, verwenden Sie dazu einen `std::stack<char>`.

Ausserdem soll das Programm sich an die Konvention für UNIX-Utilities halten:

- Wenn kein Fehler auftritt, soll das Programm gar nichts ausgeben und `0` zurückgeben.
- Bei einem Fehler soll eine Information über den Fehler nach `std::cerr` geschrieben werden. Ausserdem soll das Programm eine positive Zahl zurückgeben. An der Zahl soll zu erkennen sein, welcher der folgenden Fehler aufgetretren ist:
  - Es gibt noch ungeschlossene Klammernpaare. Am return code soll abzulesen sein, welche Art von Klammer als nächstes geschlossen werden sollte.
  - Es wurde ein falsches Klammerpaar gefunden, d.h. eine offene Klammer von Typ A wurde durch eine geschlossene Klammer von Typ B geschlossen.

Hinweise:

- Verwenden Sie die Eingabe-Funktion aus der Anfänger-Aufgabe.
- Benutzen Sie ein **`switch`**-Statement.

Übung 4 *Kalender*

Schreiben Sie eine Klasse `Date`, die ein Datum als Kombination von Tag, Monat und Jahr repräsentiert. Die Klasse soll folgende Funktionalität haben:

```
1  class Date
2  {
3    // initialisiert die Klasse mit 0 für alle Werte
4    Date();
5    // initialisiert die Klasse mit den gegebenen Werten
6    Date(int year, int month, int day);
7    // Überprüft, ob das gespeicherte Datum gültig ist
8    bool valid() const;
9    // Addiert die gegebene Zahl von Tagen zum Datum hinzu
10   void addDays(int days);
11   // Addiert die gegebene Zahl von Monaten zum Datum hinzu
12   void addMonths(int months);
13   // Addiert die gegebene Zahl von Jahren zum Datum hinzu
14   void addYears(int months);
15   // Gibt den gespeicherten Tag zurück
```

```
16    int days() const;
17    // Gibt den gespeicherten Monat zurück
18    int month() const;
19    // Gibt das gespeicherte Jahr zurück
20    int year() const;
21    // Gibt aus, wie viele Tage zwischen diesem Datum und other liegen
22    int daysAfter(const Date& other) const;
23  };
```

Hierbei ist folgendes zu beachten:

- Die Konstruktoren sollen zum Initialisieren der Member-Variablen *initializer lists*[4] verwenden.
- Die Member-Variablen sollen **`private`** sein.
- Für alle Kalenderberechnungen müssen die Schaltjahr-Regeln[5] beachtet werden. Hierbei schreiben Sie die Regeln des gregorianischen Kalenders beliebig weit in die Vergangenheit und Zukunft fort.
- Die Methoden `addDays()`, `addMonths()` und `addYears()` dürfen aus einem gültigen Datum niemals ein ungültiges machen. Das heißt insbesondere:
  - Wenn durch das Addieren von Monaten in einen Monat gewechselt wird, der weniger Tage, muss der Tag auf den letzten Tag dieses Monats korrigiert werden.
    Beispiel: 31.10.2017 → 30.11.2017.
  - Wenn das Jahr ein Schaltjahr ist und das Datum der 29.02., dann muss der Tag beim Wechsel in ein Nicht-Schaltjahr auf den 28. korrigiert werden.
- Die Funktion `daysAfter(const Date& other)` soll die *Gesamtzahl* an Tagen zwischen den beiden Daten ausgeben:

```
1  Date a(2017,11,20);
2  Date b(2016, 6,30);
3  int days = a.daysAfter(b); // 508
```

- Schreiben Sie eine Funktion `std::ostream& operator<<(std::ostream&,const Date&)`, so dass man ein Datum im Format `01.02.2017` ausgeben kann. Um die eventuell benötigten führenden Nullen bei den Monaten und Tagen zu erzeugen, können Sie `std::setfill`[6] und `std::setw`[7] verwenden.
- Die Klasse und ihre Implementierung sollen in zwei Dateien `date.hh` und `date.cc` platziert werden. Bauen Sie mit CMake ein Buildsystem, das eine Bibliothek namens `date` erzeugt, und schreiben Sie ein kleines Testprogramm, das jede Funktion von `Date` einmal testet. Dieses Programm soll gegen die Bibliothek gelinkt werden (für nähere Informationen zu CMake siehe letztes Übungsblatt).

(Fortgeschritten)

[4] https://en.wikipedia.org/wiki/Initialization_(programming)#Initializer_list
[5] https://de.wikipedia.org/wiki/Schaltjahr
[6] http://en.cppreference.com/w/cpp/io/manip/setfill
[7] http://en.cppreference.com/w/cpp/io/manip/setw