PD Dr. U. Köthe *Heidelberg, 7. November 2016*

Übungsblatt 3

Aufgabe 3.1 Gaußsche Osterregel [6 Punkte]

Erstellen Sie ein Programm, das mit dem Algorithmus von Gauß für ein gegebenes Jahr $y \geq 1583$ das Datum von Ostern (genauer: von Ostersonntag) berechnet. Dabei ist $\lfloor x/y \rfloor$ eine Division, die zur nächstkleineren ganzen Zahl abrundet (die Unterscheidung von *truncating division* und *floor division* ist hier irrelevant, weil alle Größen positiv sind). "$x \mod y$" ist eine Infix-Schreibweise für die Modulo-Operation. Eine hervorragende Erklärung, was hinter der Osterregel steckt, findet sich bei `http://www.nabkal.de/gauss.html`.

Input: Jahreszahl y.

$a \leftarrow y \mod 19$
$b \leftarrow y \mod 4$
$c \leftarrow y \mod 7$
$k \leftarrow \lfloor y/100 \rfloor$
$p \leftarrow \lfloor (8k+13)/25 \rfloor$
$q \leftarrow \lfloor k/4 \rfloor$
$m \leftarrow (15 + k - p - q) \mod 30$
$d \leftarrow (19a + m) \mod 30$
$n \leftarrow (4 + k - q) \mod 7$
$e \leftarrow (2b + 4c + 6d + n) \mod 7$
$x \leftarrow 22 + d + e$

$$z \leftarrow \begin{cases} 50 & \text{falls } x = 57 \\ 49 & \text{falls } x = 56 \text{ und } d = 28 \text{ und } a > 10 \\ x & \text{andernfalls} \end{cases}$$

Output: Ostern ist am $\begin{cases} \text{z. März} & \text{falls } z < 32 \\ \text{(z-31). April} & \text{andernfalls} \end{cases}$

Legen Sie die Datei `ostern.cpp` an und implementieren Sie obigen Algorithmus in einer Funktion `std::string easter(int year)`. Um den Typ `std::string` benutzen zu können, müssen Sie mit `#include <string>` den entprechenden Header einbinden. Die Zahl z bzw. $z - 31$ können Sie mit der Funktion `std::to_string()` in einen String umwandeln. Die Fallunterscheidungen realisieren Sie mit Hilfe der Infix-Schreibweise für Bedingungen `(condition) ? true_result : false_result`.

Testen Sie Ihre Implementation mit `assert()`-Aufrufen in der `main()`-Funktion für zehn verschiedene Jahreszahlen (`assert()` befindet sich im Header `<cassert>`, den Sie ebenfalls inkludieren müssen). Testdaten können Sie mit einem Osterrechner aus dem Internet erzeugen, z.B. `http://www.convertalot.com/easter_date_calculator.html`.

Aufgabe 3.2 Wochentagsregel [15 Punkte]

Aus der Vorlesung wissen Sie, dass man den Wochentag für ein beliebiges Datum (gegeben durch Tag d, Monat m und Jahr y) ab dem Jahre 1583 in folgenden Schritten berechnen kann:

1. Berechne den Wochentag j_1 für den 1. Januar des gegebenen Jahres:

$$\begin{aligned} z &\leftarrow y - 2001 \\ j_1 &\leftarrow (365z + \lfloor z/4 \rfloor - \lfloor z/100 \rfloor + \lfloor z/400 \rfloor) \mod 7 \end{aligned}$$

2. Berechne, der wievielte Tag im Jahr das gegebene Datum ist:

$$p \leftarrow \begin{cases} d & \text{falls } m = 1 \text{ (d.h. Januar)} \\ d + 31 & \text{falls } m = 2 \text{ (d.h. Februar)} \\ d + 59 + \lfloor (153m - 457)/5 \rfloor & \text{falls } m > 2 \text{ und } y \text{ ist kein Schaltjahr} \\ d + 60 + \lfloor (153m - 457)/5 \rfloor & \text{falls } m > 2 \text{ und } y \text{ ist Schaltjahr} \end{cases}$$

   Ein Jahr ist Schaltjahr, wenn die Jahreszahl durch 4, aber nicht durch 100 teilbar ist, oder wenn sie durch 400 teilbar ist.

3. Benutze beide Ergebnisse, um den gesuchten Wochentag zu bestimmen:

$$w \leftarrow (j_1 + p - 1) \mod 7$$

Der Wochentag w ist jetzt durch eine Zahl von 0 (Montag) bis 6 (Sonntag) kodiert. Die Notation $\lfloor a/b \rfloor$ steht dabei für eine *floor division*: wenn die Division a/b nicht exakt aufgeht, wird zur nächstkleineren ganzen Zahl abgerundet. Allerdings ist die Integer-Division in C++ als *truncating division* implementiert, d.h. positive Ergebnisse werden abgerundet, negative hingegen *auf*gerundet. Beispielsweise wird (-12)/5 bei truncating division zu -2 und bei floor division zu -3 gerundet. Wir können die Integer-Division von C++ deshalb nur nutzen, solange z nicht negativ ist, also für $y \geq 2001$.

(a) Implementieren Sie eine Funktion `int weekday2001(int d, int m, int y)`, die den obigen Algorithmus mit der truncating division realisiert. Für Fallunterscheidungen verwenden Sie die Infix-Schreibweise für Bedingungen `(condition) ? true_result : false_result`.

Testen Sie für einige Daten mit $y \geq 2001$ die Korrektheit Ihrer Funktion und überzeugen Sie sich, dass für $y < 2001$ falsche Ergebnisse herauskommen. Testdaten (also Daten, deren Wochentag bekannt ist) können Sie mit einem Wochentagsrechner aus dem Internet erzeugen, z.B. `http://www.convertalot.com/day_of_week_calculator.html` (Vorsicht: manche Wochentagsrechner im Internet geben falsche Ergebnisse!).

(b) Implementieren Sie eine Funktion `int floorDiv(int a, int b)`, die das Verhalten der floor division realisiert.

Tipp: Bei positiven Ergebnissen unterscheiden sich truncating und floor division nicht, Sie können in diesem Fall die normale C++ Integer-Division verwenden. Das geforderte Abrunden bei negativen Ergebnissen kann man durch einen Trick ebenfalls auf die truncating division zurückführen: Man berechnet mit Hilfe von truncating division den Ausdruck

$$-\frac{|a| + c}{|b|}$$

für ein geschickt gewähltes c. Überlegen Sie, welchen Wert Sie für c (in Abhängigkeit von a und b) einsetzen müssen und implementieren Sie `floorDiv()` entsprechend.

(c) Ändert man das Verhalten der Division, muss man auch eine neue Version `int floorMod(int a, int b)` der Modulo-Operation bereitstellen, damit die Invariante

```
a == floorDiv(a, b) * b + floorMod(a, b)
```

wieder gilt. Wie unterscheidet sich `floorMod()` von der `%`-Operation in C++? Implementieren Sie `floorMod()`.

(d) Benutzen Sie `floorDiv()` und `floorMod()`, um eine Funktion `int weekday(int d, int m, int y)` zu implementieren, die für alle $y \geq 1583$ funktioniert, und testen Sie diese Funktion für 16 Daten, die die verschiedenen Fälle abdecken (vor/nach 2001; Schaltjahr/kein Schaltjahr/ausfallendes Schaltjahr; verschiedene Monate, insbesondere vor/nach dem Schalttag).

(e) Wir können auch ohne `floorDiv()` und `floorMod()` auskommen, indem wir in der ersten Zeile des Algorithmus $z \leftarrow y - 1$ statt $z \leftarrow y - 2001$ schreiben. Begründen Sie, warum dies funktioniert, und implementieren Sie die Funktion `int weekday1(int d, int m, int y)`, die sich von `weekday2001()` aus (a) nur durch die erste Zeile unterscheidet. Testen Sie die Korrektheit mit denselben Daten wie in Teilaufgabe (d).

Geben Sie Ihre Lösung im File `weekday.cpp` ab, wobei die Tests mit Hilfe von `assert()` in der `main()`-Funktion durchgeführt werden. Die Antworten auf gestellte Fragen fügen Sie als Kommentare in den Code ein.

Aufgabe 3.3 Implementation der Sinusfunktion [11 Punkte]

In dieser Aufgabe wollen wir eine eigene Implementation der Sinusfunktion erstellen und sie mit der Funktion `std::sin()` aus der C++-Standardbibliothek vergleichen. Um `std::sin()` verwenden zu können, müssen Sie mit `#include <cmath>` den entprechenden Header einbinden. Informieren Sie sich in der C++-Referenz über den Inhalt dieses Headers – er enthält weitere für die Aufgabe nützliche Funktionen, z.B. `std::abs()`. Geben Sie Ihre Lösung im File `sinus.cpp` ab (die geforderten Tabellen können als Komentare in dieses File eingefügt werden).

(a) Für kleine Argumente kann man den Sinus durch die ersten Glieder seiner Taylor-Reihe approximieren. Die Approximation mit zwei Gliedern lautet

$$\sin(x) \approx \text{taylor\_sin}(x) = x - \frac{x^3}{6}$$

Implementieren Sie `taylor_sin(x)`. Berechnen Sie die Werte von `std::sin(x)` und `taylor_sin(x)` für die Winkel $5°$, $10°$, $20°$, $45°$, $90°$, $135°$, $180°$, $225°$, $270°$ sowie $315°$ und bestimmen Sie den Absolutbetrag des jeweiligen Approximationsfehlers. Beachten Sie dabei, dass Winkel stets im *Bogenmaß* an die beiden Funktionen übergeben werden müssen – Sie müssen also alle Winkel vorher aus dem Gradmaß umrechnen (z.B. durch eine Hilfsfunktion). Stellen Sie die Ergebnisse in einer Tabelle dar. Sie werden erkennen, dass die Abweichungen mit x ansteigen. Bis zu welchem Wert x_0 (auf $1°$ genau) kann man die Näherung $\text{taylor\_sin}(x)$ benutzen, wenn ein Fehler von 10^{-6} nicht überschritten werden soll?

(b) Für größere Werte von x kann man die Berechnung mit dem folgenden Additionstheorem auf kleinere Werte zurückführen:

$$\sin(x) = 3\sin(x/3) - 4\left(\sin(x/3)\right)^3$$

Implementieren Sie eine Funktion `double pump_sin(double sin_third)`, der man den Wert von $\sin(x/3)$ übergibt und die daraus mit dem Additionstheorem den Wert von $\sin(x)$ berechnet. (Tipp: Die Funktion enthält keine Aufrufe von Sinus!)

(c) Implementieren Sie eine Funktion `double my_sin(double x)`, die den Sinus rekursiv berechnet:

- Wenn $|x| \leq x_0$ (mit x_0 aus Aufgabenteil (a)): berechne das Ergebnis mittels der Näherung `taylor_sin(x)`.
- Andernfalls berechne das Ergebnis durch das Additionstheorem, also durch einen rekursiven Aufruf von `my_sin(x/3.0)` gefolgt von `pump_sin()`.

Fallunterscheidungen realisieren Sie mit Hilfe der Infix-Schreibweise für Bedingungen `(condition) ? true_result : false_result`. Bestimmen Sie wiederum den Absolutbetrag des Approximationsfehlers für die obigen Winkel.

(d) Für sehr große Werte von x sind viele rekursive Aufrufe notwendig, bis schließlich die Näherung $\mathrm{taylor\_sin}(x)$ angewendet werden kann. Das ist aufwändig und vergrößert den Fehler. Man kann diesen Aufwand vermeiden, indem man ausnutzt, dass der Sinus eine periodische Funktion ist: $\sin(x + 2k\pi) = \sin(x)$ für beliebiges ganzzahliges k. Erweitern Sie Ihre Implementation aus (b) so, dass x-Werte außerhalb des Intervalls $-\pi \leq x \leq \pi$ direkt auf dieses Intervall umgerechnet werden, indem Sie das passende k im Ausdruck $2k\pi$ bestimmen und diesen Ausdruck von x subtrahieren. Dasselbe könnten Sie mit der Funktion `std::fmod()` aus der C++-Standardbibliothek erreichen, die Sie hier aber *nicht* aufrufen (jedoch als Inspiration verwenden :-) dürfen. Der Wert von π ist in C++ in der Konstante `M_PI` gespeichert.

Testen Sie die neue Implementation ebenfalls für die gegebenen und die entsprechenden negativen Winkel.

Aufgabe 3.4 Schnelle Potenzberechnung [8 Punkte]

Bei der Berechnung von Potenzen kann man sich einige Multiplikationen sparen. Zum Beispiel kann man

$$x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$$

in drei Schritten rechnen:

$$\begin{aligned} y = x^2 &= x \cdot x \\ z = x^4 &= y \cdot y \\ x^8 &= z \cdot z \end{aligned}$$

und benötigt dann nur 3 statt 7 Multiplikationen. Allgemein lässt sich das durch folgende Rekursion ausdrücken:

$$x^n = \begin{cases} x & \text{falls } n = 1 \\ \mathrm{sq}\left(x^{n/2}\right) & \text{falls } n \text{ gerade} \\ x \cdot x^{n-1} & \text{falls } n \text{ ungerade} \end{cases}$$

Schreiben Sie eine Funktion `double power(double x, int n)`, die auf diese Weise schnell Potenzen mit `n` > 0 berechnet. Implementieren Sie eine Hilfsfunktion `sq()`

zum Quadrieren einer Zahl und verwenden Sie diese im Fall "n ist gerade". Verwenden Sie den `%`-Operator, um festzustellen, ob n eine gerade oder ungerade Zahl ist. Realisieren Sie Fallunterscheidungen mit Hilfe der Infix-Schreibweise für Bedingungen `(condition) ? true_result : false_result`.

Warum wäre es eine schlechte Idee, im Fall "n ist gerade" anstelle von $\mathrm{sq}\left(x^{n/2}\right)$ eine direkte Multiplikation $x^{n/2} \cdot x^{n/2}$ mit zwei rekursiven Aufrufen von `power()` zu verwenden? (Tipp: Probieren Sie aus, wie viele Multiplikation Sie dann insgesamt für die Berechnung von x^8 benötigen würden!)

Vergleichen Sie Ihre Ergebnisse für 8 verschiedene Testfälle mit dem Ergebnis von `std::pow()` aus der C++-Standardbibliothek. Geben Sie Ihre Lösung im File `potenz.cpp` ab. Beantworten Sie die Frage nach dem Fall "n ist gerade" in einem Kommentar.

Bitte laden Sie Ihre Lösung spätestens bis 16. November 2016, 9:00 Uhr in Moodle hoch.