Praktikum
Programmierung 3
für Technische Informatik

Konzeption:
Prof. Dr. Oliver Henkel

Durchführung:
Hendrik Plitzner

Wintersemester 2021 / 2022

Praktikumsregeln

§1 Ziel des Praktikums ist, Programmierfertigkeiten zu erlernen und durch die kontinuierliche Bearbeitung der Praktikumsaufgaben nachzuweisen. Im Praktikum herrscht deshalb Anwesenheits- und Aktivitätspflicht (praktikumsnahe Aktivität, ansonsten zählt der Termin als ein Fehltermin).

§2 Bis zu 2 Fehltermine werden toleriert und müssen nicht begründet werden. Jeder weitere Fehltermin führt unabhängig vom Fehlgrund zum Nichtbestehen des Praktikums.

§3 Die Bearbeitung der Aufgaben findet einzeln statt. Der Praktikumsbetreuer kann stichprobenweise bearbeitete Aufgaben begutachten. Erfüllt die vorgestellte Lösung die Kriterien einer durch den Praktikumsbetreuer festgelegten Mindestanforderung in nicht ausreichendem Maße, fällt die Begutachtung negativ aus.

§4 Das Praktikum ist sicher bestanden, wenn folgende Bedingungen erfüllt sind:

- Anwesenheit und Aktivität nach §1 und §2 erfüllt
- Sinnvolle und eigenständige Bearbeitung der Aufgaben
- positive Begutachtungen durch den Praktikumsbetreuer

§5 Das Praktikum wird nicht benotet und kann beliebig oft wiederholt werden.

§6 Die Aufgaben sind thematisch gruppiert sollen in dem jeweils angegebenen Zeitraum bearbeitet werden.

§7 Alle notwendigen Daten zur erfolgreichen Bearbeitung der Aufgaben sind in der jeweiligen Aufgabenstellung aufgeführt oder in Form zusätzlicher Dateien den Aufgaben beigefügt. Eine Verwendung von nicht in den Aufgabenstellungen enthaltenen Hilfsmitteln ist nur dann eine zulässige Aufgabenbearbeitung, wenn diese Hilfsmittel detailliert dargelegt, verstanden und in vollem Umfang erklärt werden können.

Kodierregeln

Dokumentation

- Jedes Programm enthält einen Kopf mit Programmnamen und Kurzbeschreibung, Autor, Datum und ggf. einer Liste der ToDos (noch zu erledigender Änderungen, Fehlerbeseitigungen, zu entfernender Warnungen etc.):

```
/*
 * ------------------------------------------------------------
 *
 * Aufgabe: xxx.cpp
 * Autor:
 *
 * kurzer Beschreibungstext
 *
 * ggf. Testfaelle/Bemerkungen/ToDos
 *
 *
 * Praktikum Programmierung 2, HS Osnabrueck
 * ------------------------------------------------------------
 */
```

- Jede Funktion/Methode erhält einen Kopf mit Namen, kurzer Beschreibung, Parameterliste und Rückgabewerte:

```
/*
 * Funktion: xxx
 * kurzer Beschreibungstext
 *
 * in|out param1 : Typ -- Beschreibung
 * in|out param2 : Typ -- Beschreibung
 * ggf. noch eine Zeile fuer Ausnahmen:
 * throws        : Typ -- Beschreibung
 *
 * return        : Typ -- Beschreibung
 *
 */
```

- Zwischen sinnvollen Abschnitten des Programms werden Kommentare zur Trennung eingefügt.
  Beispiel: Eingabe, Berechnung, Ausgabe

- Der Quellcode enthält auch in Kommentaren keine Umlaute
- Der Quellcode soll einheitlich formatiert sein. Insbesondere sind Anweisungsblöcke zwischen öffnender und schließender geschweifter Klammer geeignet einzurücken.

Variablen / Klassenattribute

- Variablen werden nicht für unterschiedliche Zwecke im Programm benutzt. Für jeden einzelnen Zweck ist eine eigene Variable zu definieren.
- Alle verwendeten Namen sind sprechend. Abkürzungen dürfen nur verwendet werden, wenn sie für den Programmierzweck eine eigenständige Bedeutung haben (z.B. physikalische Einheiten, Bezeichner in mathematischen Ausdrücken...)
- Variablen-, Attributs-, Methoden- und Funktionsnamen beginnen mit einem Kleinbuchstaben. Bei aus mehreren Wörtern zusammengesetzten Namen (nicht: zusammengesetzte Wörter wie Startpunkt, Anfangswert) werden ab dem zweiten Wort die ersten Buchstaben jeweils groß geschrieben (Beispiel: `uhrGestartet, zaehlerPunkte`)
- Nur Schleifenvariablen oder Variablen mit einer sehr kurzen Lebensspanne dürfen einfachere Namen haben
- Klassennamen beginnen mit einem Großbuchstaben. Für jede Klasse gibt es (in C++) eine Header- und eine Quellcode-Datei, deren Dateinamen mit dem Klassennamen übereinstimmen
- Konstanten werden groß geschrieben (ggf. durchgehend, mit Unterstrich als Worttrenner)

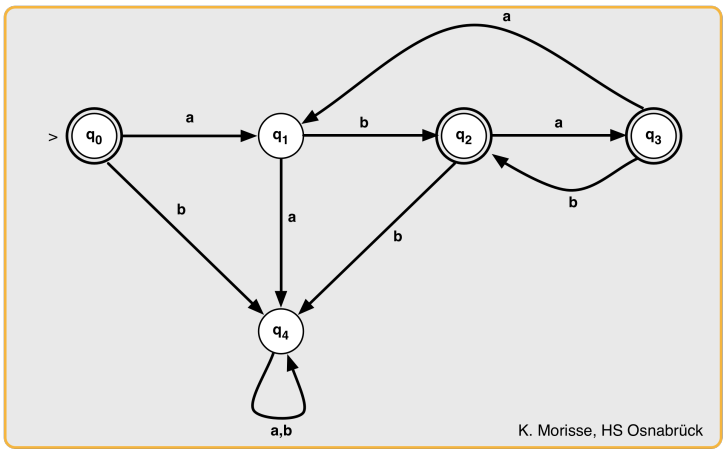
Inhaltsverzeichnis

Kapitel 1

Entwurfsmuster [2 Wochen]

1.1 Zustandsautomat

Simulieren Sie mithilfe des „State“-Entwurfsmusters den folgenden Zustandsautomaten

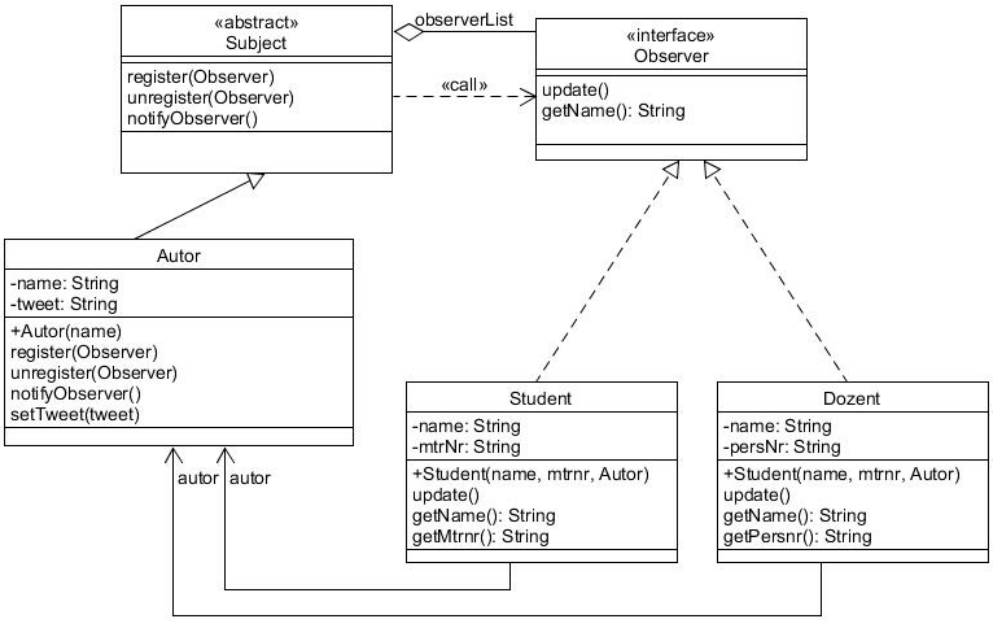


Der mit „>“ markierte Zustand ist der Anfangszustand, doppelt umkreiste Zustände sind anvisierte Endzustände. Alle anderen Zustände sind Zwischenzustände, bzw. Sackgassen.

Die Zustandsübergänge sind die angebenen **char**-Werte a oder b. Testen Sie den Automaten mit verschiedenen „Worten“, z.B. aab, aabb, aba, abb, abba, baa, bab...und bestimmen jeweils, ob ein Endzustand erreicht wird oder nicht. Welche Eigenschaft muss eine Zeichensequenz haben, damit sie vom Anfangs- in einen Endzustand gelangt?

1.2 Beobachter

Entwickeln Sie mithilfe des „Observer“-Musters folgende Anwendung



Konkrete Beobachter (Student/Dozent) können sich bei einem konkreten Subjekt (Autor) an- bzw. abmelden. Ist der Beobachter registriert, empfängt er automatisch Nachrichten, die vom Autor versendet wurden.

Kapitel 2

Container [1.5 Wochen]

2.1 Textstatistik

Schreiben Sie ein Programm `textstatistik.cpp`, das mithilfe geeigneter assoziativer Container und Algorithmen folgendes leistet: Eine einzulesende Textdatei (Testfälle sind die Datei `glocke` und `buergschaft`) soll analysiert werden.

a) Für jeden vorkommenden Buchstaben (in Kleinbuchstaben konvertiert) soll die Häufigkeit bestimmt werden.

b) Alle Worte sollen gesammelt werden und auf dem Bildschirm eine Tabelle erstellt werden, die für die Werte n=2 bis n=20 die Anzahl der Worte mit dieser Wortlänge angibt. Auf eine Benutzereingabe von n hin sollen alle Worte mit n Buchstaben ausgegeben werden.

Hinweis: Eine geschickte Containerwahl vereinfacht den Algorithmus erheblich

- Zwischen Groß- und Kleinschreibung von Worten wird nicht unterschieden
- Worte enthalten mindestens zwei Buchstaben und weder Satz- noch Sonderzeichen
- Doppelt vorkommende Worte sollen natürlich nur einmal (gespeichert und) ausgegeben werden
- Versuchen Sie, mit so wenig Schleifen/Iterationen wie möglich auszukommen

2.2 Algorithmen

Schreiben Sie Funktionen, die für eine übergebene Iteratorsequenz

- nach Eingabe einer Positionsnummer n das n-kleinste Element zurückliefert
- eine Liste zurückliefert, deren n-tes Element die Summe der ersten n Werte der übergebenen Sequenz ist
- eine Liste zurückliefert, deren n-tes Element das Produkt der ersten n Werte der übergebenen Sequenz ist

Nicht erlaubt sind Schleifen und Sortieren. Schreiben Sie ein Hauptprogramm, dass Ihre Funktionen testet. Denken Sie auch an die leere Sequenz.

Kapitel 3

funktionale Programmierung [1 Woche]

3.1 Primzahlsieb

Implementieren Sie das Primzahlsieb des Eratosthenes in C++ nach dem funktionalen Programmierparadigma:

```
67 -- Primzahlsieb
68 --   Parameter: Liste [2,3,4,...,n]
69 --   return:    Liste von Primzahlen bis n
70 sieb :: [Int] -> [Int]   -- Umklammerung [] kennzeichnet eine Liste
71 sieb (x:xs)              -- (x:xs) kennzeichnet eine Liste mit Listenkopf x
72   | (xs==[])  = []       -- [] meint die leere Liste
73   | otherwise = x: sieb [n | n <- xs, mod n x > 0]
74                          -- "|" innerhalb einer Liste meint: "für die gilt"
75                          -- "<-" meint "entnimmt aus"
76                          -- "," meint: "diejenigen Elemente, für die gilt"
77                          -- "mod n x>0" meint (in C++/Java): "n%x>0"
78
```

Die Funktion soll eine Iteratorsequenz (bidrektional) übergeben bekommen und nur durch (rekursive) Funktionsaufrufe/keine Schleifen auskommen.

3.2 Mergesort

Implementieren Sie den Mergesort-Algorithmus für integer-Werte in C++ nach dem funktionalen Programmierparadigma:

Die Funktion soll eine (bidirektional) Iteratorsequenz übergeben bekommen und nur durch (rekursive) Funktionsaufrufe/keine Schleifen auskommen.

```
92  -- Mergesort
93  merge [] ys = ys
94  merge xs [] = xs
95  merge (x:xs) (y:ys)
96    | x <= y = x:merge xs (y:ys)
97    | otherwise = y:merge (x:xs) ys
98  medianpos liste = div (length liste) 2  -- "div" meint Ganzahldivision
99  aufteilen liste = splitAt (medianpos liste) liste
100                    -- length, splitAt sind Standard-Haskellfunktionen
101 mergeSort [] = []
102 mergeSort [x] = [x]
103 mergeSort liste = merge (mergeSort links) (mergeSort rechts)
104   where (links, rechts) = aufteilen liste
105                    -- where ("wobei") ist Standard-Haskell
106
```

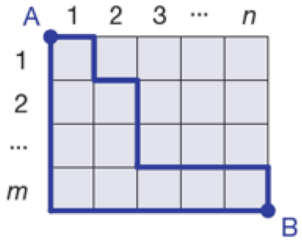
Die Funktion soll außerdem ein Sortierkriterium (kleiner, größer, Betragskleiner) übergeben bekommen. Diese sollen jeweils als Funktionszeiger und als Lambdafunktion vorliegen.

Kapitel 4

Rekursion [1.5 Wochen]

4.1 Alle Wege

Entwickeln Sie eine rekursive Funktion, die die Anzahl aller Wege entlang der Gitterlinien von der oberen linken Ecke A zur unteren rechten Ecke B bestimmt.



Beantworten Sie im Zuge der Problemlösung die Fragen:

- Welche Fälle sind Basisfälle?
- Wie sieht das rekursive Bildungsgesetz aus?

4.2 n-th Element

Bestimmen Sie aus einer Liste von Integer-Werten (Typ `list<int>`) und Eingabe einer Zahl n das n.-kleinste Listenelement, d.h. denjenigen Wert, der bei einer Sortierung der Liste an der n-ten Stelle stünde.

Es darf aber nicht sortiert werden. Stattdessen sollen Sie einen rekursiven Algorithmus nach folgender Idee implementieren: Die Liste wird rekursiv aufgeteilt in eine Liste mit Elementen, die kleiner als das erste Listenelement sind und die Liste der Elemente, die größer als das erste Listenelement (im folgenden: das Pivot-Element) sind.

Besitzt die Menge mit den kleineren Elementen genau $n - 1$ Elemente, so ist das Pivot-Element das gesuchte. Besitzt die Menge mit den kleineren Elementen mehr als $n - 1$ Elemente, so ist das gesuchte Element in dieser Menge zu finden (nämlich auch dort das n-t Kleinste), d.h. es kann rekursiv diese (kleinere) Liste nach demselben Schema aufgeteilt werden. Besitzt die Liste mit den kleineren Elementen jedoch weniger als $n-1$ Elemente, so muss die Menge mit den größeren Elementen rekursiv aufgeteilt werden.

4.3 Binomische Zahlen

Schreiben Sie ein Programm, dass nach Eingabe einer ganzen Zahl n die Binomialkoeffizienten $\binom{n}{k}$ für $k = 0$ bis $k = n$ ausgibt. Die Binomialkoeffizienten sind für natürliche Zahlen n und $k \leq n$ rekursiv definiert durch:

$$\binom{n}{0} = 1\,, \quad \binom{n}{1} = n\,, \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

und Null in allen anderen Fällen.

- Schreiben Sie dafür eine rekursive Funktion `binomi_rekursiv(n,k)`, die die Berechnung rekursiv durchführt.
- Schreiben als eine Version `binomi_Stapel(n,k)`, die die Rekursion iterativ mithilfe eines Stapelspeichers imitiert
- Schreiben Sie eine performantere iterative Variante `binomi_iterativ(n,k)`, die sukzessive die Werte in den Zeilen des Pascalschen Dreiecks berechnet, bis die Stelle (n,k) erreicht ist. Benutzen Sie zur Vereinfachung die Symmetrie $\binom{n}{k} = \binom{n}{n-k}$, um für die Berechnung stets $k \leq n - k$ annehmen zu können.

  Folgende Überlegung ist hilfreich:

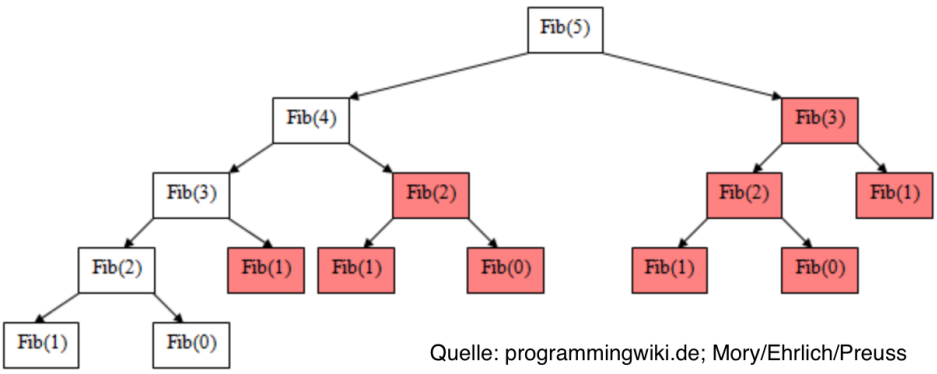
	k			
	1	1		
	1	2	1	
	1	3	3	1
n	1	4	6	4
	1	5	10	10
	1	6	15	20
	1	7	21	35
	1	8	28	56

  Der Binomialkoeffizient $\binom{n}{k}$ ergibt sich zeilenweise aus der vorhergehenden Zeile. Es müssen dabei nur für die ersten $\min(k+1, n-k+1)$ Werte ausgerechnet werden. Es genügt also, einen Zwischenspeicher für jeweils $k+1$ Werte anzulegen und dann zeilenweise aus diesen Werten die jeweiligen Werte für die nachfolgende Zeile zu berechnen. Der oben abgebildeten Skizze können Sie beispielhaft die iterative Berechnung von $\binom{8}{5}$ entnehmen.

Ihr Programm soll sicher sein, d.h. für jede numerische Eingabe korrekt arbeiten und ohne globale Variablen auskommen. Die entrekursivierte Fassung soll nach der oben beschriebenen Anleitung angefertigt werden.

Hinweis: Der größte Binomialkoeffizient, der noch in eine Variable vom Typ integer passt, ist $\binom{33}{17} = 1.166.803.110$

Bemerkung:
Die Implementation nach Art `binomi_iterativ(n,k)` nennt man dynamische Programmierung. Diese Lösungsstrategie besteht darin, zunächst Teillösungen (ggf. auch rekursiv) zu bestimmen und zwischenzuspeichern. Die Gesamtlösung wird dann aus den Teilen zusammengesetzt. Bei den in der Vorlesung vorgestellten Formen der iterativen Berechnungsmethoden der Fibonaccizahlen geschieht dies durch Wiederholung der Sequenz `g=f1+f2; f1=f2; f2=g;`, wobei die aktuellen Zwischenergebnisse nach jedem Schleifendurchlauf gerade den Werten von `f1` und `f2` bestehen und die Berechnung längs des linken (ungefärbten) Teilbaums im Rekursionsbaum stattfindet.



Quelle: programmingwiki.de; Mory/Ehrlich/Preuss

Kapitel 5

generische/generative Programmierung [1.5 Wochen]

5.1 Vektortemplates

Schreiben Sie eine Template-Bibliothek für das Rechnen mit 3-dimensionalen Vektoren, deren Einträge stets ganzzahlig sein sollen. Die Bibliothek soll mindestens die Addition, Subtraktion, sowie das Vektorprodukt enthalten und in einem Hauptprogramm getestet werden. Alle Berechnungen sollen bereits zur Compilezeit ausgefuehrt werden.

5.2 nichtnegatives Produkt

Verwenden Sie Traits, für die Prüfung der Anforderung, in einer Listenschablone `list<T>` das Produkt aller Werte durchzuführen, sofern der Elementtyp T ein nichtnegativer ganzer Zahltyp ist oder ein Gleitpunkt-Typ ist. Die Erfüllung der Anforderungen soll bereits durch den Compiler überprüft werden.

5.3 Introspektion — bedingte Kompilierung

Implementieren Sie mithilfe von Templates eine bedingte Kompilierung für ein Programm, dass abhängig davon, ob ein Container die Methode `push` oder `push_back` zur Verfügung stellt, die passende Methode aufruft.

Die Container sollen jeweils Elemente vom Typ int haben.

Kapitel 6

Weitere Aufgaben

6.1 ...können im Laufe der Vorlesung nachgereicht werden