Übung Programmieren II

Prof. Dr. Uwe Wienkop
Stand: 9. März 1999

Übersicht:

1 Einstieg Objektorientierung 2
1.1 Annuitätentilgung ☺ 2

2 Implementierung von Datentypen 3
2.1 Fifo/Stack-Implementierung 3
2.2 Listen-Implementierung ☺ 3

3 Objektorientierung in Anwendungen 4
3.1 Objektorientierte To-Do-Listenverwaltung 4
3.2 Adreßverwaltung 4

4 Operatorüberladen 5
4.1 Eigener Datentyp 'Point' 5
4.2 Implementierung des Datentyps "Rationale Zahlen" 6
4.3 Implementierung des Datentyps "String" ☺ 6

5 Projekt 7
5.1 Simulation eines Aufzugs ☺ 7
5.2 Simulation eines einfachen Prozessors 8
5.3 Simulation eines Prozeßschedulers 9

6 Klassenableitungen 10
6.1 Organisation von Fahrzeugdaten ☺ 10
6.2 Graphikobjektverwaltung 10

7 Templates 11
7.1 Fifo-Implementierung 11
7.2 Klasse für flexiblen Feldzugriff ☺ 11

8 Konzeption von größeren Programmen 12
8.1 Auftragsverwaltung 12
8.2 Programmdesign Flottenverwaltung ☺ 12

Die Übungsaufgaben sind in Klassen eingeteilt, die einzelnen Lernzielen in der Vorlesung entsprechen. Programmieren Sie daher aus jeder Klasse mindestens eine der Übungsaufgaben vollständig. Die beste Übungsaufgabe aus jedem Abschnitt ist jeweils mit einem '☺' gekennzeichnet. Sinnvoll ist auch die übrigen Übungsaufgaben zumindest „konzeptionell zu durchdenken“, d.h. daß Sie eine konkrete Vorstellung von der Lösung entwickelt haben ohne diese dann jedoch (voll) auszuprogrammieren.

---

1 Einstieg Objektorientierung

1.1 Annuitätentilgung ☺

(analog zur Übung aus Prog. I)

Durch diese Übungsaufgabe soll Bankkunden ein Tilgungplan für ein beantragtes Darlehen erstellt werden. Es wurde eine Annuitätentilgung vereinbart. Weitere Darlehenskenngrößen:

> Darlehen: 100.000 DM, Zins: 5% p.a. (eff), Tilgung: 3% p.a.
> (Bitte lassen Sie diese Größen über die Tastatur abfragen!)

Erstellen Sie jedoch im Gegensatz zur Übung aus Programmieren I nun für diese Aufgabe eine Klasse, welche die betreffenden Darlehenskenngrößen eines Kunden aufnehmen kann und dann folgende Elementfunktionen (Methoden) bereitstellt:

Konstruktor	Programmieren Sie zwei Varianten des Konstruktors, einmal mit der Möglichkeit, (Darlehens-) Parameter zu übergeben und einmal ohne Parameterübergabe (Dies soll dann die Abfrage dieser Parameter von der Tastatur zur Folge haben)
PrintTable()	Gibt eine Tabelle aus, in der jeweils in einer Zeile die folgenden Daten stehen, bis das Darlehen vollständig zurückgezahlt wurde. Lfd. Jahr | Restdarlehen | Zinsen in diesem Jahr | Tilgung in diesem Jahr
GetTotal()	Liefert den Gesamtbetrag, der zurückbezahlt werden muß (also Darlehen+Zinsen)
GetRate()	Liefert die monatliche Belastung

Hilfestellung (Annuitätentilgung)

Annuität (Jahresrate) =	Darlehen * (Zins+Tilgung) / 100 *(Diese Rate bleibt über die gesamte Laufzeit konstant!)*
Zinsen pro Jahr =	Restdarlehen * Zins/100
Tilgung pro Jahr =	Annuität – Zinsen pro Jahr

Experimente:

- Legen Sie neben dem obigen Darlehensantrag noch ein Darlehen für einen weiteren Kunden (oder für z.B. 10 Kunden [Feld!] an und experimentieren Sie mit den Zugriffsfunktionen auf die Klasse (verschiedene Aufrufe)
- Reflektieren Sie, wie Sie dies in C hätten programmieren müssen? Wie hätten Sie die Darlehensdaten der 10 Kunden abgelegt?

Lerninhalte:

- *Verwendung objektorientierter Strukturen, Erkennen (Erahnen) von Möglichkeiten des objektorientierten Ansatzes*

2 Implementierung von Datentypen

2.1 Fifo/Stack-Implementierung

Schreiben Sie ein Programm, das ein FIFO oder einen Stack für Zeichenketten der max. Länge von 30 Zeichen implementiert.

- Die zu implementierenden FIFO und Stack sollen jeweils 12 Einträge speichern können. Im Fehlerfall ist "Speicherüberlauf" bzw "Speicherunterlauf" auszugeben.
- Implementieren Sie die folgenden Funktionen:
  - int FifoIn(char *) // Schreibt die Zeichenkette in das FIFO; liefert 1 bei Erfolg
  - int FifoOut(char *) // Kopiert die Daten aus dem Fifo in den übergebenen Speicherbereich; Liefert wieder 1 bei Erfolg
  - int Push(char *) // dto.
  - int Pop(char *) // dto.

- **main (Im Hauptprogramm sollen in einer Schleife Befehle abgefragt werden)**
  - FI - Abfrage eines weiteren Textes und Einspeichern per FifoIn
  - FO - Ausgeben des nächsten Eintrags aus dem FIFO
  - SI/SO - dto. für Stackoperationen
  - E - Ende der Schleife

2.2 Listen-Implementierung ☺

Schreiben Sie ein Programm, welches einzugebende Namen in einer linearen Liste (sortiert) einfügen kann. Insgesamt sollen die folgenden Elementfunktionen bereitgestellt werden:

Konstruktor:	Erzeugen einer neuen Liste (nicht nur eines neuen Listenelements)
AddItem(Name)	Sortiertes Einfügen des Namens
First	Liefert das erste Element der Liste
Next	Liefert das jeweils nächste Element der Liste

Erstellen Sie unter Verwendung dieser Funktionen eine Liste, fügen Sie einige Namen der Liste hinzu und programmieren Sie einen Durchlauf durch die Liste bis zum Erreichen des Endes.

3 Objektorientierung in Anwendungen

Wichtige Randaspekte bei den Aufgaben aus diesem Abschnitt

- Achten Sie auf eine "saubere" Programmierung, d.h. insbesondere
  - strukturierte Programmierung
  - Trennung von "allgemeinen" Abfragefunktionen mit der Datenverwaltung (lineare Liste) und Datenablagefunktionen.
  - Konsequentes Einhalten von Einrückregeln, welche Sie auch immer wählen
- Versuchen Sie, die Datenhaltung mit ihren Zugriffsfunktionen in einer eigenen Datei und entsprechende Prototypen in einer .h – Datei zu implementieren
- Verwenden Sie für Eingabe und Ausgabe bitte die C++ cout und cin – Streams!

3.1 Objektorientierte To-Do-Listenverwaltung

Modifizieren Sie Ihr Programm zur To-Do-Listenverwaltung aus Programmieren I gemäß den Kenntnissen, die Sie jetzt zur objektorientierten Programmierung erworben haben, d.h. trennen Sie "allgemeine" Abfragefunktionen von der Datenverwaltung (lineare Liste) und und den Datenablagefunktionen. Gruppieren Sie die Funktionen in geeignete Klassen.

3.2 Adreßverwaltung

Konzipieren und implementieren Sie ein Programm für die Verwaltung von persönlichen Adreßdaten (z.B. Name, Vorname, Telefonnummer, Straße, Ort und PLZ) und evtl. weitere zugehörige Daten. Hierbei sind die folgenden Aspekte zu beachten:

- Der Programmstart erfolgt mit einem optionalen Parameter, der eine Datei mit schon gespeicherten Adreßdaten beschreibt. Falls dieser Parameter angegeben wird, ist diese Datei einzulesen und in einer linearen Liste abzuspeichern.
- Danach geht das Programm in einen Kommandomodus. Nun sollen die folgenden Befehle zur Verfügung stehen:
  - 'E' <RET> – Eingabe eines neuen Datensatzes. Die einzelnen Elemente des Datensatzes sollen von der Tastatur abgefragt werden. Anschließend wird der Datensatz in einer linearen Liste abgespeichert.
  - 'S <Name>' <RET>. Nun soll der Adreßdatensatz zu dem angegebenen Namen gesucht und auf dem Bildschirm angezeigt werden.
  - 'L <Name>' <RET>. Auch hier soll zunächst der Datensatz gesucht und angezeigt werden; Anschließend soll abgefragt werden, ob dies der für das Löschen gewünschte Datensatz ist? Bei Verneinung der Frage soll weitergesucht werden, ansonsten ist der Datensatz aus der Liste zu löschen.
  - 'W <Dateiname>' <RET>. Der gesamte Inhalt der Liste ist in die angegebene Datei zu sichern; Dateiformat s.u.
  - 'A' <RET>. Der Inhalt der linearen Liste wird datensatzweise auf dem Bildschirm ausgegeben und jeweils auf eine Bestätigung gewartet.
  - 'X' <RET>. Beendigung des Programms. Sollten noch nicht alle Daten abgespeichert worden sein, so ist noch eine Bestätigung einzuholen bzw. sind die Daten entsprechend abzuspeichern.

Dateiaufbau

- Die Adreßdaten befinden sich in einer Textdatei, wobei in jeder Zeile genau ein Datensatz steht
- Die einzelnen Elemente eines Datensatzes sind durch " " (z.B. "Neue Str.") gekennzeichnet, so daß jedes Element eindeutig zugeordnet werden kann.

Lerninhalte:

- *Dynamische Speicheranforderung*
- *Lineare Liste mit einigen Zugriffsfunktionen*
- *Umgang/Konzeption/Implementierung von etwas größeren Programmieraufgaben (mehrere Programmdateien)*
- *Entwurf einer einfachen Kommandoschnittstelle*

4 Operatorüberladen

Die Lernschwerpunkte der Aufgaben in diesem Abschnitt sind der Mechanismus, um Operatoren zu überladen, Unterschiede zwischen Element- und Nicht-Elementfunktionen beim Überladen, die besondere Bedeutung des Copy-Konstruktors und auch das Überladen der Wertzuweisung.

4.1 Eigener Datentyp 'Point'

(Klausuraufgabe WS98/99, Zeitvorstellung: ca. 35 Min.)

a) Klasse CPoint

Bitte implementieren Sie (ohne die Verwendung der Klasse Vector der Standardbibliothek) eine neue Klasse **`CPoint`**, welche die Handhabung von Koordinaten vereinfachen soll. Beim Erzeugen eines Objekts dieser Klasse soll angegeben werden können, welche **Dimension** dieser Punkt hat; Beispiel:

```
CPoint  x(1);      // 1-dimensional, entspricht: x hat nur eine x-Komponente
CPoint  p(2);      // 2-dimensional, entspricht: p hat x und y-Komponenten
CPoint  q(3);      // 3-dimensional, entspricht: q hat x,y und z-Komponenten
```
usw.

Um die Dimension beliebig halten zu können, sollen die Komponenten in einem Feld abgelegt werden, welches zum Zeitpunkt der Objekterzeugung angefordert wird. Der Datentyp der Komponenten sei **double**.

Implementieren Sie eine Klasse CPoint, die dieses leistet und folgende weitere Eigenschaften besitzt:

- Der **Zugriff auf eine Komponente** soll wie ein Zugriff auf ein Feld geschehen; Beispiele:
  `p[0]` ~ x-Komponente des Punkts p
  `q[1]` ~ y-Komponente des Punkts q
  `q[2]` ~ z-Komponente des Punkts q
- Falls versucht wird, auf eine Komponente außerhalb der angegebenen Dimension zuzugreifen, so ist die Ausnahme "**`OutOfDimension`**" zu generieren.
- Über obigen Komponentenzugriffsmechanismus sollen auch die Komponentenwerte gesetzt oder gelesen werden können, so daß ***zum Beispiel*** folgende Zugriffe möglich sind:
  ```
  p[0]  = 100.0; // x-Komponente von p auf 100.0 setzen
  p[1]  =   0.0; // y-Komponente von p auf 0.0 setzen
  x[0] +=  p[0]; // x-Komponente von p zur x-Komponente von x hinzuaddieren
  ```

  ***Hinweis**: Bitte überlegen Sie, welche **eine** Zugriffsart auf Elemente **alle** diese Zugriffe erlaubt!*
  Geben Sie NICHT für jede Möglichkeit eine eigene Überladung an!!!

- Die **Konstruktion** eines 'Punkt-Objekts' aus einem anderen 'Punkt-Objekt' soll möglich sein
  Beispiel:
```
CPoint  q(3);
CPoint  p = q;
```
- Die **Zuweisung** eines Punkt-Objekts an ein (evtl. anderes) Punkt-Objekt soll nur dann gelingen, wenn die Dimension des Ausdrucks auf der rechten Seite der Zuweisung kleiner oder gleich der Dimension des Objekts auf der linken Seite ist. Falls die Dimension der rechten Seite echt kleiner ist, so werden nur die vorhandenen Komponenten kopiert; Beispiel:
```
CPoint  p1(2), p2(2), q(3);
   // ...
p1 = p2;  // o.k.
q  = p1;  // o.k. es werden nur die in auch in p1 enthaltenen Komponenten von q überschrieben.
p2 = q;   // Fehler, da die Dimension von q drei, von p2 jedoch nur zwei beträgt!
```
- Im Fehlerfall ist die Ausnahme "**`DimensionMismatch`**" zu generieren.
- Die Ausnahmen sollen aus Komfortgründen sowohl einzeln als auch gesammelt als "**`CPointErrors`**" gefangen werden können.

b) Main – Programm

Schreiben Sie ein einfaches main-Programm, welches nur die Aufgabe hat, die beiden Ausnahmen DimensionMismatch und OutOfDimension der Klasse CPoint hervorzurufen und abzufangen. Es ist ausreichend, wenn das main-Programm als Fehlerbehandlung lediglich eine Fehlermeldung auf dem Bildschirm ausgibt.

4.2 Implementierung des Datentyps "Rationale Zahlen"

Implementieren Sie den in Vorlesung ansatzweise vorgestellten Datentyp "Rationale Zahlen". Dies betrifft insbesondere:

- vollständiger Satz an Basisoperationen +, - (unär, binär), *, /
- Einige Vergleichsoperatoren
- gemischte Arithmetik mit Integerzahlen

4.3 Implementierung des Datentyps "String" ☺

- Die Klasse "string" soll einen privaten Zeiger auf einen dynamischen Speicher besitzen.
- Es sollen Variablen für Stringlänge und Stringpuffergröße angelegt werden. Der Stringpuffer soll bei der Konstruktion 15 Zeichen größer sein als die Stringlänge selbst.
- Sie soll drei Konstruktoren besitzen: String (void), String(const char*), String (const char) diese besetzen den Speicherplatz entsprechend
- Schreiben Sie drei Zuweisungsoperatoren operator=(String), operator=(const char*) und operator=(const char) sowie einen copy-Konstruktor String(const String&).
- Überladen Sie die Operatoren << und >> für Strings.
- Überladen Sie die Operatoren + (Hintereinanderschreiben zweier Strings) und += (String1+=String2 bedeutet, daß hinter String1 noch String2 angehängt werden soll).
- Fügen Sie eine Einfügefunktion hinzu, die an der Stelle x im Stringspeicher die ersten n Buchstaben eines zweiten Strings einfügt.
- Überladen Sie den [ ]-Operator, um auf die einzelnen Zeichen eines String-Objekts genau zugreifen zu können
- Überladen Sie für die Klasse String die Vergleichsoperatoren.

– Überladen Sie den ( )-Operator so, daß Sie ihn mit (pos,len) aufrufen können und er dann einen Teilstring erzeugt, der an der Position pos beginnt und len Zeichen lang ist.

5 Projekt

Die hier aufgeführten Aufgaben sind umfangreicher und es bietet sich hier an, die verschiedenen Aufgaben jeweils auf mehrere Klassen zu verteilen. Die Identifizierung der Klassen, die Aufteilung der Funktionen auf Klassen und die Kommunikation zwischen den Klassen ist das Lernziel dieser Projekt-Übungsaufgaben!

Wichtig ist es bei diesen Projektaufgaben jedoch, (mindestens) eine der Aufgaben wirklich vollständig durchprogrammiert und nicht nur 'entworfen' zu haben.

Gehen Sie hierbei wie folgt vor:

- **Überlegen Sie zuerst**, welche Aktionen Sie in **Klassen** zusammenfassen wollen (z.B. eigene Klassen für ...)
- Überlegen Sie, wie Sie den **Simulationsvorgang** gestalten wollen
- Listen Sie die Beziehungen/**Verknüpfungen zwischen den Klassen** auf.
- Implementieren Sie den Simulator derart, daß das eigentliche Simulationsverfahren gut gekapselt ist (eigene Klasse). Es soll leicht möglich sein, dieses Verfahren zu modifizieren.
- Implementieren Sie die einzelnen Klassen des Simulators in eigenen Dateien!

5.1 Simulation eines Aufzugs ☺

Konzipieren (1) und **implementieren (2)** Sie einen Simulator für eine Aufzugssteuerung, etwa wie man sie im FH-Gebäude A (für **einen** Aufzug) findet. Der Simulator soll alle wesentlichen Vorgänge eines Aufzugs simulieren können und eine Bewertung der Aufzugssteuerung erlauben.

Vorgänge des Aufzugs:

- Anforderungsknopf in einer (oder mehreren) Ebene(n) gedrückt
- Tür auf/zu
- Lichtschranke frei
- Personen treten ein und drücken den Knopf für das gewünschte Fahrziel
- Fahrstuhl setzt sich gemäß bekannter Strategie in Bewegung
- Personen verlassen den Aufzug
- Gewichtsüberwachung, d.h. Alarmglocke, wenn zu viele Personen eintreten. Keine Fahrt antreten, Türen nicht schließen, bis Gewicht wieder reduziert wurde

Bewertungskriterien für eine Aufzugssteuerung
(Für diese Kriterien soll der Simulator entsprechende Daten liefern)

- **Durchsatz** (jede Aktion (Tür auf/zu, Fahrt des Aufzugs zum nächsten Stockwerk, Lichtschranke frei, auf frei warten, etc.) kostet Zeit. Gesucht ist die **durchschnittliche Zeit**, vom Eintreffen von Personen vor dem Aufzug bis sie dann wieder den Fahrstuhl auf der gewünschten Etage verlassen.
- **Akzeptanz**; Menschen tendieren dazu, nach einer bestimmten Zeit des Wartens vor dem Aufzug, ohne daß dieser eingetroffen ist, das Warten abzubrechen und die Treppe zu nehmen. Gesucht ist die **Abbrechquote** (Abbruch zur Gesamtzahl der Personen)

Hinweise

- **Überlegen Sie zuerst**, welche Aktionen Sie in **Klassen** zusammenfassen wollen (z.B. eigene Klassen für Personen, Aufzug, Simulation, ...)
- Überlegen Sie, wie Sie den **Simulationsvorgang** gestalten wollen, z.B. zufälliges Bestimmen einer Etage, auf der eine neue Person den Aufzug anfordern soll. Diese fordert den Aufzug an, Fahrstuhlsimulation geht einen Schritt weiter, falls Fahrstuhl ankommt, dann tritt die Person ein, falls Überlast, etc.)
- Listen Sie die Beziehungen/**Verknüpfungen zwischen den Klassen** auf.
- Implementieren Sie den Simulator derart, daß der eigentliche Aufzug mit seinen Sensoren, Aktoren und insbesondere der Fahrstrategie gut gekapselt ist (eigene Klasse). Es soll leicht möglich sein, mit alternativen Fahrstrategien zu experimentieren.
- Bei der Simulation können Sie so vorgehen, daß Sie die kleinste Zeiteinheit bestimmen, die im (Aufzug-) System vorkommen kann. Diese können Sie als Simulationstakt wählen und alle anderen Aktionen relativ dazu wählen bzw. abtesten.
- Implementieren Sie die einzelnen Teilaufgaben des Simulators in eigenen Dateien!

5.2 Simulation eines einfachen Prozessors

Entwickeln und implementieren Sie einen Simulator, der einen einfachen Prozessor bei der Abarbeitung von Assemblerprogrammen nachbildet. Der Prozessor habe 8 Register mit den Namen a, b, c, ... h. Diese Register können jeweils 8 Bit-Werte aufnehmen. Der Prozessor kann auf einen Programmspeicher der Größe 1000 Befehle und auf einen Datenspeicher der Größe 1000 Bytes zugreifen.

Der Prozessor soll die folgenden Befehle kennen:

Befehl	**Beispiel**	**Kommentar**
load <reg>, <Datenspeicheradr.>	load a,100	Liest den Wert aus der Hauptspeicherzelle und speichert ihn im Register ab
store <reg>, <Datenspeicheradr>	store a,100	Speichert den Wer aus Register a in der Hauptspeicherzelle 100 ab. Der Wert im Register bleibt erhalten
mov <reg>, <reg>	mov a,c	Kopiert den Registerinhalt von 'a' nach Register 'c'
mvi <reg>, Konstante	mvi a, 4	Initialisiert das Register 'a' mit der nachfolgenden Konstante, hier 4 (mvi – move immediate)
add <ziel-reg>, <reg>, <reg>	add a, b, c	Addiert die Inhalte der Register 'b' und 'c' und speichert das Ergebnis im Zielregister ab, hier Register 'a' Ein evtl. Überlauf ist in einem zusätzlichen carry-Flag zu speichern Beeinflußt ebenfalls das zero-Flag
addc <ziel-reg>, <reg>, <reg>	addc a, b, c	Analog zur add, jedoch wird hier zum Ergebnis der Addition von b und c noch der Wert des carry-Flags hinzuaddiert.
inc <reg>	inc a	Inkrementiere den Inhalt von Register 'a'
dec <reg>	dec a	Dekrementiere den Inhalt von Register 'a' Von inc, dec und add wird ein zero-Flag beeinflußt. Dieses Flag zeigt an, ob das Ergebnis dieses Befehls gleich oder ungleich Null war. Dies kann z.B. bei Sprungbefehlen abgetestet werden.
jump <befehlsnummer>	jump 20	Springt zum Befehl an Befehlszeile 20
jnz <befehlsnummer>	jnz 20	Springt zum Befehl an Befehlszeile 20 falls das zero-Flag nicht gesetzt war (jump on not zero)
jnc <befehlsnummer>	jnc 20	dto, falls das Carry-Flag nicht gesetzt war (jump on not carry)

in <Datenspeicheradr>	in 100	liest eine Zahl von der Tastatur (Eingabeaufforderung) und speichert diese an der angegebenen Datenspeicheradresse ab, hier 100
out <Datenspeicheradr>	out 100	gibt die an der angegebenen Datenspeicheradresse abgespeicherte Zahl auf dem Bildschirm aus *Hinweis: Diese beiden letzten Befehle (in/out) finden sich in dieser Form nicht auf realen Prozessoren, da Ein-/ Ausgabebefehle meist sehr komplex sind und viele Befehle zu ihrer Realisierung benötigen.*

Beispielprogramm:

```
in    100         // Zahl abfragen
in    101         // zweite Zahl abfragen
load  a, 100      // erste Zahl ins Reg. a laden
load  b, 101      // zweite Zahl ins Reg. b laden
add   c, a, b     // a und b add, Ergebnis in c speichern
store c, 102      // Wert von c im Datenspeicher abspeichern
out   102         // ... und ausgeben
jmp   0           // Verfahren (immer) wiederholen; Sprung zur ersten
                  // Befehlszeile
```

Die Programme für diesen Prozessor sollen nun in einer Datei abgelegt sein. Der Prozessorsimulator wird dann mit diesem Dateinamen gestartet, z.B. prozessorsim <datei>

und anschließend soll das in dieser Datei befindliche Programm abgearbeitet werden.

Simulationsaufgaben:

- Schreiben Sie ein Assemblerprogramm für die Addition zweier 32 Bit-Zahlen (also 4x8 Bit)
- Schreiben Sie ein Programm für die Addition zweier beliebig langer Zahlen (abfragen)

5.3 Simulation eines Prozeßschedulers

Entwickeln und implementieren Sie einen Simulator, der das Verteilverhalten eines Schedulers als Kernstück eines Betriebssystems untersuchen hilft. An diesen Scheduler kommen immer wieder (Ankunftsrate einstellbar gestalten!) neue Prozesse (Rechenaufträge) heran. Jeder Auftrag besitzt eine Ankunftszeit (Zeitpunkt der Erzeugung) und einen bekannten Rechenzeitbedarf (Rechenzeitsspanne ebenfalls einstellbar gestalten!)

Unser Scheduler soll diesen Prozesse nun nach dem sogenannten Round-Robin-Verfahren Rechenzeit zuweisen. Bei dem Round-Robin-Verfahren darf jeder Prozeß ein bestimmtes Zeitquantum rechnen und muß dann einem anderen Prozeß den Prozessor überlassen. So geht es mit der Prozessorzuweisung reihum (round-robin), bis die Prozesse vollständig abgearbeitet sind. In dieser Simulation braucht natürlich kein Programm für die zu simulierenden Prozesse abgearbeitet werden, sondern es reicht hier aus, wenn die Restrechenzeit einfach um das zur Verfügung stehende Zeitquantum reduziert wird!

Für die Auswertung des Verfahrens sind die folgenden Daten interessant:

- Durchschnittliche Antwortrate (Antwortrate = Rechenzeitbedarf des Prozesses geteilt durch die Zeit, die der Prozeß tatsächlich im System verbracht hat)
- Maximale Anzahl der wartenden Prozesse
- Periodische „Schnappschüsse“ bei der Simulation mit gerade aktuellen Daten wie Anzahl wartende Prozesse, aktuelle Antwortrate, etc.

6 Klassenableitungen

Wesentliches Lernziel in diesem Abschnitt ist das Kennenlernen des Ableitungsmechanismus und der daraus resultierenden programmiertechnischen Konsequenzen (z.B. Wahl von Zugriffsschutzmechanismen, virtuelle Funktionen sowie die Verwaltung von polymorphen Objekten; Stichwort: Polymorphie funktioniert in C++ nur über Zeiger!)

6.1 Organisation von Fahrzeugdaten ☺

Gesucht ist für eine Stelle wie z.B. das Kraftfahrtbundesamt eine neue, übersichtliche und leicht erweiterbare Organisation der verschiedenen Fahrzeugdaten.

Für die unterschiedlichen Fahrzeugarten (PKWs, Motorrädern und LKWs) sind jeweils unterschiedliche Daten zu erfassen. Auch die Weiterbearbeitung der Daten (z.B. Steuerberechnung) erfolgt nach unterschiedlichen Kriterien. Zu erfassende und zu speichernde Daten:

- Für alle Fahrzeugtypen
  Kennzeichen, Jahr der Erstzulassung
- PKWs, zusätzlich:
  Hubraum, Leistung, Schadstoffklasse [0-schadstoffarm, 1-normal, 2-Diesel]
- Motorräder, zusätzlich:
  Hubraum
- LKWs, zusätzlich:
  Anzahl der Achsen, Zuladung in t

Entgegen dem alten System möchte man sich bei der reinen Verwaltung der Daten (z.B. Datenablage, Suche nach Fahrzeugen gemäß Kennzeichen, etc. Steuerberechnung und TÜV-Anmahnung) nicht mehr mit unterschiedlichen Fahrzeugtypen "herumschlagen".

Implementieren Sie oben gefordertes System mit Hilfe von abgeleiteten C++-Klassen. Bieten Sie für die Auswertung/Weiterbearbeitung der Daten folgende Funktionen an:

- Suchen nach einem Fahrzeug anhand dem eingegebenen Kennzeichen und Ausgeben aller betreffenden Fahrzeugdaten
- Ausgeben aller Daten von allen gespeicherten Fahrzeugdaten
- Berechnung der Steuerschuld für 1) ein angegebenes Kennzeichen und 2) für alle gespeicherten Fahrzeuge gemäß folgenden Formeln:
  - PKW: (Hubraum+99) / 100 * 10 DM * (Schadstoffklasse+1)
  - Motorrad: (Hubraum+99) / 100 * 20 DM
  - LKWs: Zuladung*100 DM/t
- Auflisten der TÜV-Fälligkeit für alle Fahrzeuge in einem anzugebenden Jahr.
  (Sie können hier davon ausgehen, daß die erste TÜV-Untersuchung nach drei Jahren zu erfolgen hat und anschließend alle zwei Jahre eine Untersuchung fällig ist)

6.2 Graphikobjektverwaltung

(Klausuraufgabe WS98/99, Zeitvorstellung: ca.35 Min.)

Entwerfen Sie die Datenhaltungskomponente für ein Graphikprogramm. Das Graphikprogramm soll in der ersten Ausbaustufe lediglich die Figuren Linie und Kreis kennen. Für die eigentliche Darstellung auf dem Bildschirm stehen aus einer Graphikbibliothek folgende Funktionen zur Verfügung:

```
DrawLine(int x1, int y1, int x2, int y2, int LineColor);
            // Linie von x1,y1 nach x2,y2 mit Farbe LineColor zeichnen
DrawCircle(int x, int y, int radius, int LineColor, int FillColor);
            // Kreis um x,y mit Radius in der Farbe LineColor zeichnen und diesen mit der Farbe
            // FillColor ausfüllen
```

Entwerfen Sie entsprechende Klassen für die Verwaltung der genannten Figurtypen Linie und Kreis unter Hinzunahme einer gemeinsamen Basisklasse derart, daß alle Figuren z.B. in **einem gemeinsamen Feld** abgelegt werden können.

Berücksichtigen Sie beim Entwurf, daß alle Variablen in den Klassen so **lokal und geschützt** (private oder protected) wie möglich gehalten werden sollen.

Als **Zugriffsfunktionen** auf die Klassen werden jeweils folgende Funktionen benötigt:

- Konstruktion des jeweiligen Objekts unter Angabe aller Parameter
- Destruktion des Objekts
- Funktion 'GetParams': Auslesen aller Parameter eines Objekts über Call-by-reference-Mechanismus
- Funktion 'Plot': Aufruf der entsprechenden Graphikbibliotheksfunktion zur Ausgabe der Daten auf dem Bildschirm

Aufgaben:

a) Geben Sie von der *Basisklasse* und von der *Klasse Kreis* die **vollständige Klassendefinition** an

b) Erstellen Sie ein Mini-Hauptprogramm, in dem Sie von jeder der zwei Klassen ein Objekt erzeugen und dieses so in **ein gemeinsames** Feld eintragen, daß später ein bequemer Durchlauf durch das Feld z.B. mittels einer for-Schleife möglich wird, etwa in der Art:
"feld[0] = Kreis"
"feld[1] = Rechteck"
"feld[2] = Kreis"

c) Programmieren Sie die for-Schleife, mit der Sie alle Objekte auf dem Bildschirm anzeigen lassen (Aufruf der Plot-Funktionen)

7 Templates

7.1 Fifo-Implementierung

Erweitern Sie die Fifo-Realisierung aus Aufgabe 2.1 derart, daß bei der Definition Fifos beliebigen Typs erzeugt werden können.

7.2 Klasse für flexiblen Feldzugriff ☺

Entwerfen und implementieren Sie eine Templateklasse für einen abstrakten Datentyp „Feld“ von einem anzugebenden Typ, bei dem jedoch keine Feldgrößen angegeben werden. Diese Feldgrößen sollen zur Laufzeit automatisch erkannt und angepaßt werden.

Die Basis für diesen Datentyp sei eine lineare Liste. Bei jedem Listenelement sei neben dem Wert auch der Index gespeicht, um den es sich handelt. Existiert bei einem Zugriff bereits ein solches Element, so wird es verwendet, ansonsten wird es erst neu angelegt. Der Zugriff soll über die folgenden Methoden geschehen:

[]	Indexzugriff, falls ein Index noch nicht existiert, wird er hierbei automatisch erzeugt!
del(Index)	Löschen des Feldelements mit dem angegebenen Index

8 Konzeption von größeren Programmen

Im Gegensatz zu den Projektaufgaben – wo die Aufgaben tatsächlich programmiert werden sollten – steht in diesem Kapitel der *Entwurf bzw. die Konzeption* im Vordergrund. Hier soll geübt werden, relativ schnell zu einer komplexeren Aufgabenstellung eine (später realisierbare) Lösung zu *entwerfen*.

8.1 Auftragsverwaltung

Die Fa. Commerz & Co plant die Einführung eines neuen Auftragverwaltungssystems. Dieses System soll folgendes leisten:

1. Entgegennahme von Kundenbestellungen in der Form: <Kunde k> bestellt <Produkt y> der <Fa. z> jeweils <m – Stück>. Jede Bestellung erhält eine vom System vergebene eindeutige Bezeichnung <Bestellzeichen b>.
2. Entgegennahme von bestellten Warenlieferungen in der Form <Fa. z> liefert <Produkt y> in <Menge-m> für <Bestellzeichen b> zum <Einzelpreis p>
3. Entgegennahme von Lieferzusagen in gleicher Form, jedoch ergänzt um eine Datumsangabe
4. Einmal täglich sollen die Bestellungen nach Firmen sortiert abgerufen werden können.
5. Für jeden Kunden sollen die Liefertermine – sofern verfügbar – abrufbar sein
6. Es sollen die Kunden ermittelt werden können, die noch auf Warenlieferungen warten (sortiert nach Bestelldatum)
7. Wenn ein Kunde seine Waren abholt, dann ist eine entsprechende Rechnung zu drucken und der Auftragsdatensatz kann gelöscht werden

Hinweise:

- Bitte implementieren Sie Datumseinträge, Bestellungen, etc. durch entsprechende Klassen.
- Ein Datensatz kann Element mehrerer sortierter Listen sein!

- Bitte überlegen Sie sich zuerst, welche Teilaufgaben gelöst werden müssen; Versuchen Sie zu modularisieren. Versuchen Sie auch, die Abhängigkeiten zwischen den verschiedenen Datensätzen zu erkennen.

8.2 Programmdesign Flottenverwaltung ☺

(Klausuraufgabe SS98, Zeitvorstellung: ca. 50 Min.)

! *Bei dieser Aufgabe ist nur an wenigen Stellen tatsächlich etwas voll zu kodieren.*
Bitte lesen Sie die Aufgabenstellung aufmerksam durch!!! !

Die Firma TransSea plant für ihre Fährschiffe ein neues Buchungs- und Informationssystem einzuführen. Hierbei sind folgende Aspekte zu berücksichtigen:

- Da die Firma stark expandiert, ist die Anzahl der Schiffe, für die das Buchungssystem auszulegen ist, variabel zu halten.
- Jedes Schiff hat einen eindeutigen Namen [string] und eine von den anderen Schiffen unabhängige Auslegung hinsichtlich der Ladekapazität, gemessen in Gesamtladelänge [m], Gesamtladegewicht [kg] und Ge-

samtanzahl der transportierbaren Personen. Keine dieser drei Größen darf beim Beladen überschritten werden.

- Bei der Belegung eines Schiffs mit Fahrzeugen soll zwischen PKWs, Motorrädern und Bussen unterschieden werden. Für das Buchungssystem (und nachgeschaltete Programme) werden die folgenden Fahrzeugdaten benötigt:
  - PKWs: Kennzeichen, Länge, Höhe, Gewicht und Anzahl der Insassen
  - Motorräder: Kennzeichen und Anzahl der Personen; Für Motorräder wird pauschal eine Länge von 1m (wegen der Möglichkeit zum Querstellen) und ein Gewicht von 100kg angenommen.
  - Busse: Länge, Gewicht, Kennzeichen, Anzahl der Passagiere und Anzahl der Achsen
- Auch die Berechnung der Fährtarifs soll nach unterschiedlichen Kriterien erfolgen:
  - PKWs: Länge*100 + 50 DM/Person
  - Motorräder: 250 DM + 50 DM/Person
  - Busse: 2000 DM + 30 DM/Person

Aufgaben der Buchungskomponente des Systems (b)

1. Aufnahme eines neuen Transportwunsches für ein Schiff, welches über seinen Namen adressiert wird.
2. Dieser Transportwunsch muß dann gemäß den obigen Beladungskriterien überprüft werden; Ungültige Buchungen (aufgrund von Überschreitungen) der Grenzwerte sollen zurückgewiesen werden.
3. Ein Transportwunsch kann unter Angabe von Schiffsnamen und KFZ-Kennzeichen storniert werden

Aufgaben der Informationskomponente des Systems (i)

1. Einrichten eines neuen Schiffs, d.h. auch Abfrage der entsprechenden Beladungsgrenzwerte
2. Eine schiffsweise Aufstellung der aktuellen Belegungsstände, d.h. jeweils die prozentuale Gesamtbelegung nach Längen, Personen und Gewicht
3. Ausgeben des Gesamtumsatzes, der für ein – über den Namen – benanntes Schiff, derzeit erzielt wird.

Implementierungsangaben

Verwenden Sie für die Verwaltung eines Schiffs eine Klasse CSchiff und für die Verwaltung aller Schiffe eine Klasse CFlotte. CFlotte soll dabei eine lineare Liste aller Schiffe und CSchiff eine lineare Liste aller Fahrzeuge enthalten. Sie dürfen dabei auf eine Bibliothek mit der Implementierung einer Liste zurückgreifen. Diese Bibliothek bietet nachfolgende Zugriffsfunktionen an. Verwenden Sie ausschließlich diese bei der Realisierung der untenstehenden Aufgaben:

```
template <class T> CList {
          // ...
public:
          CList();            // Erzeugt eine leere Liste
          ~CList();           // Löscht die Liste ab einem gegebenen Index
          AddItem(T &Data);   // Fügt ein neues Datum vor dem aktuellen Listenelement ein
          RemoveData();       // Löscht das aktuelle Listenelement
          CList *Begin();     // Liefert einen Zeiger auf das erste Listenelement
          CList *Next();      // Liefert einen Zeiger auf das dem aktuellen Element nachfolgende
                                 Listenelement, NULL falls das Listenende erreicht wurde!
          T *Data();          // Liefert einen Zeiger auf die Datumskomponente des aktuellen
                                 Listenelements
};
```

Ihre Aufgaben:

1. Geben Sie für die obige Aufgabenstellung die benötigten **Klassendeklarationen** an. Verwenden Sie bei den Fahrzeugen den C++-Ableitungsmechanismus! Geben Sie bei den Fahrzeugen **nur die Basisklasse** und **die**

Klasse für die PKWs an! Achten Sie bei den Deklarationen auf die Erfüllung der obigen Aufgabenstellungen! Verwenden Sie bitte verständliche Variablen- und Funktionsnamen an, und geben Sie in Kommentaren kurz an, auf welche der obigen Aufgabenstellungen sich diese Funktion bezieht, Beispiel: "// b2", d.h. diese Funktion bezieht sich auf Aufgabe 2 der Buchungskomponente, " // i1" entsprechend auf die 1. Aufgabe der Informationskomponente.

2. Geben Sie zusätzlich nur für die PKW-Klasse auch noch die **Funktionsdefinitionen** an.

3. Geben Sie die **Definition** der Funktion zur *Bestimmung des Gesamtumsatzes des **aktuellen** Schiffs* an, also etwas wie "`double CSchiff::Umsatz()`". Die gewünschten Informationen sollen zur Laufzeit dieser Funktion aus den in den Fahrzeugen gespeicherten Informationen bestimmt werden – sind also sonst nicht verfügbar!