5.3.V Datenstrukturen II/Arrays von Records – Versuch

5.3.V.1 Kundendatenverwaltung

Sie sollen ein C-Programm `person3.c` schreiben, das mehrere Kunden (=Personen) verwalten kann. Das Programm soll folgenden Dialog anzeigen (Benutzereingaben fett):

Terminal

```
schueler@debian964:~$ person3
----------------------------
Kundendaten
----------------------------
<0> Ende
<1> Alle Daten ausgeben
<2> Neuen Datensatz eingeben
----------------------------
Ihre Auswahl: 1
Alle Daten ausgeben:
0 Meier, Hans, 19801224
1 Mueller, Egon, 19790228
2 Schulze, Hermann, 19290102
3 Lehmann, Friedrich, 19520814
----------------------------
Kundendaten
----------------------------
<0> Ende
<1> Alle Daten ausgeben
<2> Neuen Datensatz eingeben
----------------------------
Ihre Auswahl: 2
Datensatz eingeben:
Nachname: Pieper
Vorname: Gustaf
Geburtsdatum: 19821124
----------------------------
Kundendaten
----------------------------
<0> Ende
<1> Alle Daten ausgeben
<2> Neuen Datensatz eingeben
Ihre Auswahl: 0
----------------------------
Ende.
```

5.3.V.2 Planung Menüstruktur

Hier haben wir (wieder?) ein Programm, dessen Steuerung weitgehend vom Benutzer vorgegeben wird. Der Benutzer legt selbst fest, welches Unterprogramm als nächstes aufgerufen wird. Dazu bekommt er eine Auswahl, ein so genanntes Menü, angezeigt. Nach der Wahl kommt er in ein Unterprogramm mit einem neuen Dialog, in dem er seine Eingaben tätigen muss. Ist das Unterprogramm fertig, landet der Benutzer wieder im Menü. Diese Struktur kann man in einem Diagramm abbilden. Abbildung 1 zeigt ein solches Diagramm; leider gibt es keine gebräuchliche Normung für diesen Zweck, so dass man sich mit Elementen aus Flussdiagrammen aushelfen muss.

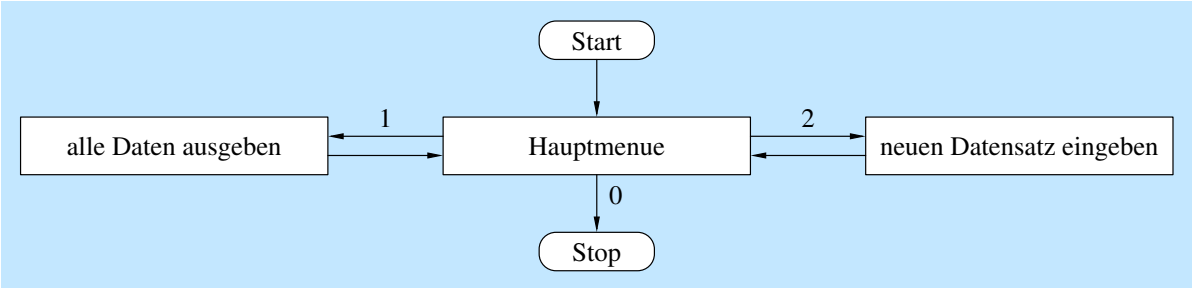

Abbildung 1: Menüstruktur

5.3.V.3 Planung Programmstruktur

Nun soll die Programmstruktur nach Top-Down-Art entworfen werden. Dazu eignen sich Struktogramm und Jackson-Diagramm sehr gut, das Flussdiagramm eignet sich weniger. Im Struktogramm fängt man mit einem leeren Rechteck an (Abbildung 2 links). Das Programm wird we-

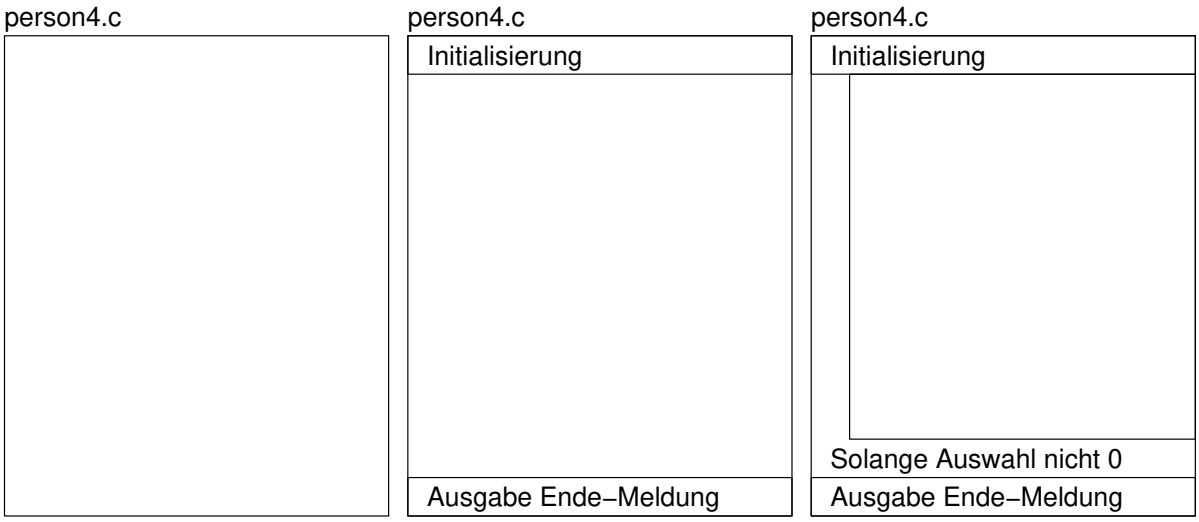

Abbildung 2: Programmstruktur, Schritte 1–3

sentlich durch das Hauptmenü bestimmt, in dem man nach Aufruf eines Unterprogramms *immer wieder* landet. Das klingt nach einer Schleife. Das Menü wird mindestens einmal angezeigt, also sollte es eine *Fußgesteuerte Schleife* sein (Abbildung 2 rechts). Vor der Schleife gibt es möglicherweise eine Initialisierung, dahinter die Ausgabe einer Ende-Meldung. So ist die Schleife in eine *Sequenz* eingebettet (Abbildung 2 Mitte).

Innerhalb der Schleife muss *zuerst* das Menü angezeigt werden, anschließend erfolgt die Eingabe der Auswahl und die Verarbeitung der Auswahl (Abbildung 3 links). Also gibt es innerhalb der Schleife eine *Sequenz*. Die Verarbeitung der Auswahl hängt davon ab, was ausgewählt wurde. Also besteht sie aus einer *Mehrfachauswahl* (auch genannt Fallunterscheidung, Abbildung 3 Mitte). In zwei der Fälle wiederum ist der *Aufruf* je eines Unterprogramms nötig (Abbildung 3 rechts). Danach muss für jedes der Unterprogramme ein eigenes Struktogramm erstellt werden.

5.3.V.4 Planung Datenstruktur

In diesem Programm müssen *viele* Kunden verwaltet werden. Da bietet es sich an, ein Array `ktabelle` anzulegen, dessen Elemente Records sind, also ein *Array von Records*:

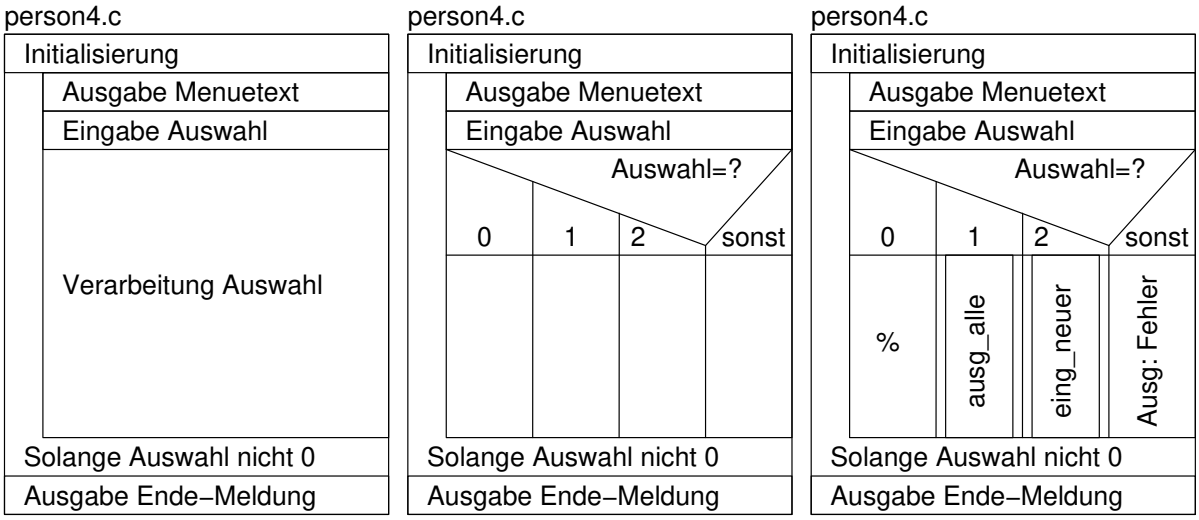

Abbildung 3: Programmstruktur, Schritte 4–6

```
1 #define TABMAX 20
2    struct persontyp ktabelle[TABMAX];
```

Nun können in `ktabelle` maximal 20 Kunden gespeichert werden. Mit `ktabelle[0]` greift man auf den ersten möglichen, mit `ktabelle[19]` auf den letzten möglichen Kunden zu. Nun kann es sein, dass zu Beginn noch keine Kunden eingetragen sind. Es ist zwar möglich, bis zu 20 Kunden einzutragen; aber die aktuelle Anzahl von Kunden ist null. Wie kann man das im Programm festhalten? Da gibt es mehrere Möglichkeiten (die Aufzählung ist nicht vollständig):

a) Man speichert die aktuelle Anzahl in einer Variablen:

```
1    unsigned int aktanz=0;
```

Alle gültigen Kunden haben nun einen Index zwischen null und `aktanz-1`.

b) Ebenso, aber man macht aus `aktanz` und `ktabelle` ein neues Record:

```
1    struct ktabelle_gesamttyp
2    {
3       struct persontyp ktabelle[TABMAX];
4       unsigned int aktanz;
5    } ktabelle_gesamt;
```

Das ist eine fast schon elegante Lösung; nur ist damit der Zugriff auf einzelne Komponenten eines Kunden sehr umständlich: `ktabelle_gesamt.ktabelle[0].gebdat=20000101;`

c) Man definiert einen Element-Inhalt, der als Terminator dient. Wenn ein Kunde z.B. das Geburtsdatum 31.12.2099 hat, dann soll dieser Kunde kein echter Kunde, sondern ein Terminator sein. Hinter diesem Kunden befindet sich kein weiterer Kunde.

d) Jeder Kunde bekommt eine eigene Komponente `int letzter_kunde`, die ihn als letzten Kunden in der Tabelle auszeichnet. Ist ihr Inhalt ungleich null, befindet sich hinter diesem Kunden kein weiterer.

e) Jeder Kunde bekommt eine eigene Komponente `int ist_gueltig`. Nur Kunden bei, denen `int ist_gueltig` ungleich null ist, sind gültig. In der Tabelle dürfen nun gültige und

ungültige Kunden in beliebiger Reihenfolge stehen. Das macht das Löschen von Kunden sehr einfach und schnell.

Es wird entschieden, die erste Lösung (getrennte Variable `aktanz`) zu verwenden.