Einführung in die Programmierung

Dateien

1. Arten von Dateien

Viele Programmiersprachen unterscheiden zwei Arten der Dateien: die Textdateien und die binären Dateien. Grundsätzlich sind alle Dateien selbstverständlich binär kodiert.

Die Textdateien haben aber eine besondere Stellung in Betriebssystemen. Sie enthalten Steuerungssymbole CR/LF (neue Zeile), dadurch ist es möglich, solche Dateien mit sehr einfachen Mitteln auf dem Bildschirm zeilenweise anzuzeigen und zu verarbeiten. Alle Daten in solchen Dateien stellen die Symbole dar (Buchstaben, Zahlen, spezielle Symbole). Die Zahlen können in solchen Dateien selbstverständlich auch gespeichert werden, aber bevor man sie verarbeitet (addiert, subtarhiert), muss man sie erkennen und in binäre Form umwandeln, z.B. ASCII-Symbol für 3 hat den Code 51 = 00110011, und die Zahl 3 sieht in der binären Darstellung so aus: 00000011.

Beim Schreiben der Textdateien werden spezielle Steuerungssymbole (CR/LF) mitgespeichert, die beim Auslesen erkannt und für Formatierung verwendet werden. Das tun die Ein- und Ausgabe-Operatoren für Textdateien.

Die Ein- und Ausgabe-Operatoren für binäre Dateien schreiben und zeigen Information zeichenweise so wie sie sind. Die Symbole CR/LF werden in dem Fall nicht für Steuerung verwendet. Der Unterschied zwischen den Textdateien und den binären Dateien liegt in den Operatoren, die für Schreiben/Lesen verwendet werden. Daraus folgt, dass man nur eine Art der Operatoren für eine bestimmte Datei verwenden soll, sonst werden die Daten wahrscheinlich falsch dargestellt.

Vorsicht: die oben beschriebenen Textdateien nicht mit Word/Writer-Dateien (.DOC, .DOCX, .ODT) verwechseln. Die letzten speichern viel mehr Steuerungssymbole für Darstellung des Textes und sollen eher zu binären Dateien zugeordnet werden.

Dateien können allgemein in einem von mehreren Modi geöffnet werden: zum Schreiben (Write), zum Hinzufügen (Append), zum Lesen (Read), zum Erstellen (Create).

2. Textdateien

Die Lese- oder Schreibvorgänge in Textdateien laufen bis das Symbol für die neue Zeile kommt ("\n"). Deswegen kann der gelesene oder geschrieben Block bei jedem Vorgang unterschiedliche Länge haben. Man sagt, dass die Operatoren für Textdateien das Symbol für die neue Zeile erkennen, oder unterscheiden, oder berücksichtigen.

Textdateien eignen sich für eine einfache Textverarbeitung. In solchen Dateien werden nicht alle ASCII-Symbole verwendet, nur die sogenannten abbildbaren Symbole (die auf der Tastatur vorhanden sind).

Beispielweise sind die Konfigurationsdateien für Linux fast immer Textdateien, und man kann sie mit sehr einfachen Mitteln verarbeiten.

```
VAR Zeile: string = 'abc';
    Q: string;
    D: FILE OF string;  // file handler, versuchen Sie auch mit D: FILE OF TEXT;
BEGIN
  Assign (D, 'daten.txt');     // Handler mit Datei verbinden
  ReWrite (D);                 // Datei zum Schreiben öffnen
  REPEAT
    Write ('... Eine Zeile eingeben: ');
    ReadLn (Zeile);    // Lesen von der Tastatur, versuchen Sie mit Read (Zeile)
    Write (D, Zeile);  // In die Datei schreiben, WRITE, nicht WRITELN
    Write ('Noch mal ? (j/n) ');
    Readln(Q);
  UNTIL Q <> 'j';
  Close (D);  //  Wie wichtig ist diese Anweisung hier?
  Reset (D);           // Datei zum Lesen öffnen
  REPEAT
    Read (D, Zeile);   // Lesen aus der Datei, READ, nicht READLN
    WriteLn (Zeile);   // Versuchen Sie mit Write (Zeile);
  UNTIL EOF (D);
  Close (D);
END.

Ablauf:
... Eine Zeile eingeben: erste zeile
Noch mal ? (j/n) j
... Eine Zeile eingeben: zweite zeile
Noch mal ? (j/n) j
... Eine Zeile eingeben: und dritte zeile
Noch mal ? (j/n) n
erste zeile
zweite zeile
und dritte zeile

Fazit: Zeilen werden so geschrieben, dass sie beim Lesen wieder erkannt werden.
```

```
package main
import ( "fmt"; "os"; "io" )

func main() {
  var s string
  var q string = "j"
  f, e := os.Create("daten.txt")  //  Achtung: Rückgabe von zwei Werten!
  for q == "j" {
    fmt.Print("... Eine Zeile eingeben: "); fmt.Scanln(&s)
    fmt.Fprintln(f,s)             // Versuchen Sie hier mit fmt.Fprint(f,s)
    fmt.Print("Noch mal ? (j/n) "); fmt.Scanln(&q)
  }
  f.Close()
  f, e = os.Open("daten.txt")
  for true {
    _, e = fmt.Fscanln(f, &s) // Erster Rückgabewert wird weiter nicht verwendet
    if e == io.EOF { break }  // End Of File
    fmt.Println(s)       // Wie sieht dann die Ausgabe aus mit fmt.Fprint(f,s) oben?
  }
  f.Close()
}
```

```
Ablauf:
... Eine Zeile eingeben: erste_zeile
Noch mal ? (j/n) j
... Eine Zeile eingeben: zweite_zeile
Noch mal ? (j/n) j
... Eine Zeile eingeben: und_dritte_zeile
Noch mal ? (j/n) n
erste_zeile
zweite_zeile
und_dritte_zeile

Fazit: Zeilen werden so geschrieben, dass sie beim Lesen wieder erkannt werden.
Problem: Leerzeichen innerhalb der Zeichenkette.
Suchen Sie einen Ersatz für fmt.Scanln(), damit die Leerzeichen innerhalb einer Zei-
chenkette berücksichtigt werden, wie im Pascal.
```

Aufgabe 1. Schreiben Sie ein Programm in Pascal und in Go, wo Sie die Zeichenketten (Zeilen) von der Tastatur auslesen und in einer Textdatei speichern. Das alles passiert, solange eine Zeichenkette in der ersten Position keinen Punkt hat. Die Zeichenkette mit dem Punkt wird in der Textdatei nicht gespeichert. Um einen Punkt in der ersten Position zu erkennen, brauchen Sie wahrscheinlich Funktion copy() in Pascal und Slice-Verarbeitung in Go, oder Erfahrungen aus dem UB_05. Inhalt der Textdatei soll auf dem Bildschirm nicht angezeigt werden. Den Schreibvorgang kann man hier genau so organisieren, wie es in obigen Beispielen steht.

Aufgabe 2. Schreiben Sie ein Programm in Pascal und in Go, dass die Zeilen aus der Textdatei (s. Aufgabe 1) ausliest. Nur die Zeilen werden auf dem Bildschirm angezeigt, die ein im Programm vordefiniertes Muster enthalten. Um ein Muster in der Zeile zu erkennen, brauchen Sie wahrscheinlich Funktion pos() in Pascal und strings.Index() in Go. Den Lesevorgang kann man hier genau so organisieren, wie es in obigen Beispielen steht.

3. Binäre Dateien

Bei binären Dateien ist es wichtig anzugeben, wie viel Byte man auf einmal lesen oder schreiben möchte. Somit ist das Ende des Lese- oder Schreibvorgangs nicht das Symbol "\n" (neue Zeile), sondern es geht immer um die Größe des Blocks (in Bytes), der gelesen oder geschrieben wird. Das Symbol für die neue Zeile kann in diesen Dateien selbstverständlich auch vorkommen, aber die Operatoren für binäre Dateien berücksichtigen es nicht, das ist ein ganz normales Symbol für sie.

Die binären Dateien eignen sich für Speicherung der aggregierten Daten, Prozessorbefehle, Bildverarbeitung, wo wirklich alle ASCII-Symbole vorhanden sein können.

```
VAR Zeile: string = '';
    B: byte;
    D: FILE OF BYTE;      // Datei wird byteweise gelesen oder geschrieben
BEGIN
  Assign (D, 'daten.txt');   // Handler mit Datei verbinden
  Reset (D);                 // Datei zum Lesen öffnen
  REPEAT
    Read (D, B);      // Ein Byte aus der Datei lesen – byte ist eine Zahl!
    Write (chr(B));   // Versuchen Sie mit Write (B);
    Zeile := Zeile + chr(B);   // Alles in eine Zeichenkette konkatenieren
  UNTIL EOF (D);
  Close (D);
  Writeln (Zeile);
END.
```

```
package main
import ( "fmt"; "os"; "io" )

func main() {
  var s, q string = "", "j"
  var b []byte
  var e error
  h, _ := os.OpenFile("daten.txt", os.O_APPEND, 0755)
  for q == "j" {
    fmt.Print("... Eine Zeile eingeben: "); fmt.Scanln(&s)
    b = []byte (s)  // Konvertierung "string to slice"
    h.Write(b)      // Alle Bytes schreiben
    fmt.Print("Noch mal ? (j/n) "); fmt.Scanln(&q)
  }
  h.Close()
  h, e = os.OpenFile("daten.txt", os.O_RDONLY, 0755)
  s = ""
  b = []byte {'x'}  // b besteht nur aus einem Element, byteweise lesen
  for {
    _, e = h.Read(b);  // Nur 1 Byte lesen
    if e == io.EOF { break }
    s = s + string(b)  // Neues Byte in die Zeichenkette einfügen
  }
  fmt.Println(s)
  h.Close()
}

Ablauf:
... Eine Zeile eingeben: aaaaa
Noch mal ? (j/n) j
... Eine Zeile eingeben: bbbbbbb
Noch mal ? (j/n) j
... Eine Zeile eingeben: cccccc
Noch mal ? (j/n) n
aaaaabbbbbbccccccc

/* Man kann auch blockweise lesen, z.B. 3 Byte auf einmal */

func main() {
  var s string = ""
  var n int
  var b []byte = []byte {'x','x','x'}  // Slice hat 3 Byte, genau so viel wird gelesen
  var e error
  h, _ := os.OpenFile("daten.txt", os.O_CREATE, 0755)
  h.WriteString("\nIch weiss nicht, was soll es bedeuten")  // "\n" ist neue Zeile
  h.WriteString("\nJ. W. von Goethe")                       // Versuchen Sie ohne "\n"
  h.Close()
  h, e = os.OpenFile("daten.txt", os.O_RDONLY, 0755)
  for {
    n, e = h.Read(b);         // n Byte wurde tatsächlich in b gelesen, e ist Error
    if n<3 { b = b[ :n] }     // Eine Korrektur kann am EOF notwendig sein
    s = s + string(b)         // Konvertierung und Konkatenieren
    if e == io.EOF { break }  // Schleife verlassen, wenn EOF
  }
  fmt.Println(s)
  h.Close()
}
```

Aufgabe 3. *Schreiben Sie ein Programm in Pascal und in Go, das auf den Aufgaben 7 und 8 (UB_05) basiert. Sie lesen mehrmals die Datensätze über Artikel von der Tastatur ein und speichern diese Daten (Records) in einer binären Datei ab.*
Das Schreiben von Datensätzen, die einen Verbund darstellen, wird mit oben beschriebenen Mitteln in Go wahrscheinlich nicht klappen. Um eine Struktur (struct) in der binären Datei zu speichern, braucht man in Go das Paket "encoding/gob" und die Funktion gob.NewEncoder(f) — f ist Dateihandler. Diese Funktion liefert einen Writer-Objekt zurück, der imstande ist, die ganze Struktur auf einmal in der binären Datei zu speichern. Das Writer-Objekt (z.B. wrt) wird so aufgerufen: wrt.Encode(s), s ist hier die zu speichernde Struktur (Verbund).

Aufgabe 4. *Schreiben Sie ein Programm in Pascal und in Go, das die vorige Aufgabe 3 ergänzt. Die Datensätze werden aus der binären Datei ausgelesen und auf dem Bildschirm angezeigt. Das Lesen von Datensätzen, die einen Verbund darstellen, wird mit oben beschriebenen Mitteln in Go wahrscheinlich nicht klappen. Um eine Struktur (struct) aus der binären Datei zu lesen, braucht man in Go das Paket "encoding/gob" und die Funktion gob.NewDecoder(f) — f ist Dateihandler. Diese Funktion liefert einen Reader-Objekt zurück, der imstande ist, die ganze Struktur auf einmal aus der binären Datei zu lesen. Das Reader-Objekt (z.B. rdr) wird so aufgerufen: rdr.Decode(&s), s ist hier die Struktur (Verbund).*

Aufgabe 5. *Schreiben Sie ein Programm in Pascal und in Go, um Textdateien und binäre Dateien zu testen. Schreiben Sie Daten in eine Datei, als ob es eine Textdatei wäre, und lesen Sie die Daten aus, als ob es eine binäre Datei wäre. Und umgekehrt. Sie sollten beobachten, dass die geschriebenen Daten nicht immer mit den gelesenen übereinstimmen.*