Teil IX : Sequentielles I/O, Dateien

1. **Problem und Konzepte**
2. **Modula-2 Lösungen**

© K. Murmann, H. Neumann, Fakultät für Informatik, Universität Ulm, 2001

1. Problem und Konzepte

- Problem(e)
- Programminteraktion
- sequentielle Datenströme

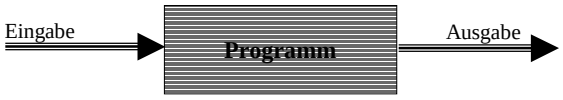
Problem(e)

- Ein Programm produziert zur Laufzeit Daten und verwaltet sie im Hauptspeicher. Sofern sie nicht rechtzeitig auf einem externen Medium (als Datei) gespeichert werden, gehen sie spätestens verloren nach Beendigung des Programms.

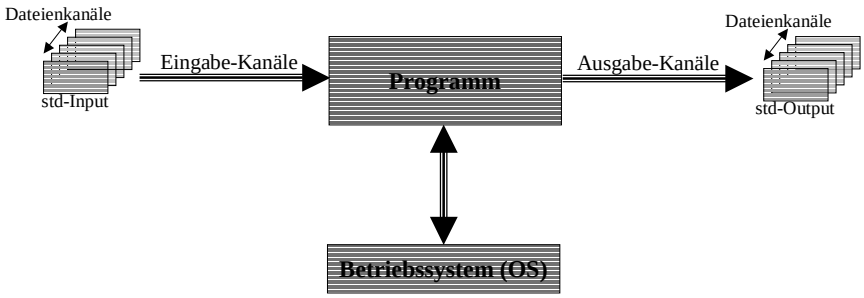
- Ich will nicht die Eingabedaten über die Konsole „hineinfüttern“, insbesondere dann, wenn bei Programm-Testläufen immer wieder die selben Daten einzugeben sind (womöglich viele). Ich will die Eingabedaten aus einer Datei auf einem externen Medium einlesen lassen.

- Aber wie?
  Die Konzeption der Dateneingabe und -ausgabe (input und output, also I/O) ist in vielen Programmiersprachen sehr verschieden.
  Der Sprachumfang von Modula-2 enthält überhaupt keine Anweisungen für I/O (und ist damit unabhängig vom Betriebssystem zu realisieren!).

Programminteraktion



genauer:



- Unter UNIX werden I/O-Geräte wie spezielle Dateien verwaltet, die reservierte File-Deskriptoren zugewiesen bekommen:
  - stdin: Standard-Eingabe über Tastatur (Deskriptor: 0)
  - stdout: Standard-Ausgabe zum Monitor (Deskriptor: 1)
  - stderr: Standard-Fehlerstrom zum Monitor (Deskriptor: 2)

Sequentielle Datenströme

- Die große Mehrzahl aller Ein- und Ausgabeoperationen kann als sequentiell angesehen werden. Die Struktur der Daten entspricht derjenigen einer Sequenz („sequence“) / eines Stroms („stream“).
- Sequenzen mit ihren Grundoperationen lassen sich als ADTs spezifizieren (wie bereits schon stack und queue).
  Operationen:
  - read/write: Lesen/Schreiben von Sequenzen
  - skip: „Ansteuern“ einer impliziten Position
  - show: Anzeige des Elements an dieser (impliziten) Position
  - reset: Zurücksetzen der Position auf 1. Element
  - eof: Position ist des letzte Element der Sequenz

2. Modula-2 Lösungen

- Modularisierung und Bibliothekslösung
- Bibliothekslösung zur Kommandozeilenverarbeitung

Modularisierung und Bibliothekslösung

➢ In Bibliotheken werden Module bereitgestellt, die zu der vorhandenen Umgebung (OS) „passen“ (deswegen Ulm‘s Modula-2 Library). Soll Modula-2 also auf eine andere Plattform portiert werden, so müssen die Module in diesen Bibliotheken angepasst werden.

Beispiele für Bibliotheken:

genaue Informationen unter: http://www.mathematik.uni-ulm.de/modula/man/

- ♠ InOut:
  formatierte I/O von INTEGERs, CARDINALs, CHARs, STRINGs über stdin bzw. stdout
- ♠ RealInOut:
  formatierte I/O von REALs über stdin bzw. stdout
- ♠ FtdIo:
  formatierte I/O über Dateien; basiert/greift zu auf StdIo;
  der „stream“ wird mittels eines Parameters angegeben (stdin, stdout, stderr sind Standard-“streams“); Funktionsumfang entsprechend InOut.

- ♠ StdIo:
  Schnittstellenmodul für Basisoperationen zur I/O = Ein-/Ausgabe = Lesen/Schreiben von Dateien.
  - Opaker Datentyp FILE („stream“, „sequence“)
  - MODE: jeder Kanal erhält (access-)Attribut (read,write,append)
  - Zugriffsoperationen zu:
    - Öffnen/Schließen
    - Datenmanipulation, Zustandsänderung
    - Zustandsabfrage
- ♠ FILES
- ♠ SysRead
- ♠ SysWrite
- ♠ SysExit

Bibliothekslösung zur Kommandozeilenverarbeitung

- Datenversorgung
  Bisher: Programme können über Standardeingabe bzw -ausgabe Daten ein- bzw. auslesen.
  Mechanismen:
  - Tastatureingabe bzw. Monitorausgabe
  - Umlenken: fn < eingabe bzw. fn > ausgabe
  - Pipe: ... | ...

  Ziel: Ein-/Ausgabe für ein Programm über Dateien, deren Namen beim Aufruf des Programms (in der Kommandozeile) mit angegeben werden, d.h. das Programm muss in der Lage sein, im Programmaufruf (= Kommando in shell) Argumente zu verarbeiten. Hierbei orientiert man sich an der Struktur von UNIX-Kommandos.
- Struktur von UNIX-Kommandos:
  Kommando Optionen Ausdruck Dateiname(n)
  Bsp: cp fn1 fn2
  grep „Sally Smith“ people

- Standardprozeduren zur Kommandozeilenverarbeitung, also zum Lesen von Kommandozeilen incl. ihrer Argumente, mit Modula-2 Programmen:

  genaue Informationen unter: http://www.mathematik.uni-ulm.de/modula/man/

  - Modul ARGC (muss nicht explizit importiert werden):
    - Prozedur ARGC:
      liefert die Anzahl der Kommandozeilenargumente incl. Kommando-name
    - Prozedur ARGV:
      speichert einzelnes Kommandozeilenargument (0 ... ARGC-1)
  - Modul Arguments
    Prozeduren für Kommandozeilenverarbeitung

- Beispiele:
  - Einlesen einer Datei:
    - Einlesen des Dateinamens von stdin
    - Öffnen der Datei zum Lesen
      falls Datei vorhanden, dann zeichenweises Einlesen (bis Feof) und zeichenweise Ausgabe auf stdout (Echo)
      falls Datei nicht vorhanden, dann Fehlermeldung und Abbruch.
  - Formatierte Ausgabe auf eine Datei:
    - Einlesen des Dateinamens von stdin
    - Öffnen der Datei zum Schreiben
      falls Datei vorhanden, dann Nachfrage ob überschreiben (falls N, dann Abbruch (exit); falls Y weiter)
      falls Datei nicht vorhanden, dann neue datei öffnen.
    - Text von stdin einlesen und formatiert in Datei schreiben
  - Modula-2 Programm, das den Inhalt einer vorhandenen Datei <fn1> in eine (neue) Datei <fn2> kopiert (→ implementiert UNIX-Kommando cp <fn1> <fn2>)

Beispiel: Einlesen einer Datei

```
MODULE FileIn;
  FROM InOut   IMPORT WriteString, WriteLn, Write, ReadString;
  FROM StdIO   IMPORT FILE, MODE, Feof, Fopen, Fclose, Fgetc;
  FROM FtdIO   IMPORT FwriteString, FwriteLn;
  FROM SysExit IMPORT Exit;

CONST
  STRLEN  = 256;

VAR
  FileName    :  ARRAY [0..STRLEN] OF CHAR;
  FilePointer :  FILE;
  ch          :  CHAR;

BEGIN
  WriteString("Bitte Dateinamen eingeben :  ");
  ReadString(FileName);

....
```

Datei auswählen

Beispiel: Einlesen einer Datei (cont'd)

```
....

IF (NOT Fopen(FilePointer, FileName, read, TRUE)) THEN
  WriteString(FileName);
  WriteString(" doesn't exist! "); WriteLn;
  Exit(1)
END;

WHILE (NOT Feof(FilePointer) AND Fgetc(ch, FilePointer)) DO
  Write(ch)
END;

IF (NOT Fclose(FilePointer)) THEN
  WriteString("Can't close ");
  WriteString(FileName);
  Exit(1)
END

END FileIn.
```

Ausstieg falls Datei nicht existiert

Einlesen und Ausgabe auf stdio

Ausstieg falls Datei sich nicht schließen lässt

Beispiel: Modula-2 Kopierprogramm

```
MODULE FileCopy;
FROM Arguments IMPORT InitArgs, GetArg, AllArgs, Usage;
FROM StdIO     IMPORT FILE, MODE, Fopen, Feof, Fclose,
                      Fgetc, Fputc, stdin, stdout, stderr;
FROM FtdIO     IMPORT FwriteString, FwriteLn;
FROM SysExit   IMPORT Exit;

CONST
  MAXARGS =   2;
  STRLEN  = 256;

VAR
  args       :  ARRAY[0..1],[0..STRLEN] OF CHAR;
  argc       :  CARDINAL;
  from, to   :  FILE;
  answer, ch :  CHAR;
```

Beispiel: Modula-2 Kopierprogramm (cont'd)

```
BEGIN
  InitArgs("SourceFile DestinationFile");

  argc :=  0;
  WHILE (argc < MAXARGS) AND GetArg(args[argc]) DO
    INC(argc)
  END;

  AllArgs; (* ... noch Argumente auf der Kommandozeile uebrig ? *)

  (* -- erstmal alle Fehler pruefen *)

  IF argc < 2 THEN
    Usage
  END;

  (* -- jetzt gilt argc = 2 *)
  (*    echo arguments ...  *)
  FOR argc := 0 TO MAXARGS-1 DO
    FwriteString(stdout,args[argc])
  END;
```

Beispiel: Modula-2 Kopierprogramm (cont'd)

```
 FwriteLn(stdout);

 IF (NOT Fopen(from, args[0], read, TRUE)) THEN
   FwriteString(stderr, "Can't open ");
   FwriteString(stderr, args[0]); FwriteLn(stderr);
   Exit(1)
 END;
```

Ausstieg falls sich die Quell-Datei nicht öffnen lässt

```
 IF Fopen(to,args[1], read, TRUE) THEN
   FwriteString(stdout, args[1]);
   FwriteString(stdout, " already exists / overwrite (y/n)? ");
   IF Fgetc(answer, stdin) AND (answer <> 'y') THEN
     Exit(2)
   END;
```

Ausstieg falls die Ziel-Datei nicht überschrieben werden darf

```
   (* -- return-Zeichen ueberlesen *)
   IF Fgetc(ch, stdin) THEN
   END;
```

Beispiel: Modula-2 Kopierprogramm (cont'd)

```
 (* -- Zieldatei wieder schliessen *)
    IF (NOT Fclose(to)) THEN
      FwriteString(stderr, "Error in Close"); FwriteLn(stderr);
      Exit(3)
    END
```

Ausstieg falls sich die Ziel-Datei nicht schließen lässt

```
  END;
  IF (NOT Fopen(to, args[1], write, TRUE)) THEN
    FwriteString(stderr, "Can't open ");
    FwriteString(stderr, args[1]); FwriteLn(stderr);
    Exit(2)
  END;
```

Ausstieg falls die Ziel-Datei sich nicht öffnen lässt

```
  (* -- jetzt kann kopiert werden *)
  WHILE (NOT Feof(from)) AND Fgetc(ch, from) AND Fputc(ch, to) DO
    (* --> Fputc(ch, to) macht die Kopierarbeit *)
  END

  (* -- schliessen der Dateien ist hier unnoetig ... ! *)
END FileCopy.
```

Σ

- Die **Ein-/Ausgabe** (I/O) von Daten erfolgt standardmäßig über die Konsole (Tastatur/Monitor). Der Sprachumfang einer Programmiersprache umfasst oft gar keine I/O-Befehle, da diese von der jeweiligen Rechnerarchitektur und dem Betriebssystem abhängen, die Sprache jedoch plattformunabhängig konzipiert ist.
- Das Problem wird gelöst durch **Bereitstellung einer Bibliothek** mit verschiedenen I/O-Modulen, deren Prozeduren zur genutzten Rechner-Plattform passen und verschiedenste Operationen mit sequentiellen Datenströmen (streams) ermöglichen, z.B. die Speicherung von Daten in einer Datei auf einem Massenspeicher (z.B. Festplatte).
- Beim **Aufruf eines Programmes** in einer Kommandozeile des Betriebssystems (bzw. dessen shell) sollen **zusätzliche Parameter** angegeben werden können und auch die Namen von Eingabe- und Ausgabedateien. Auch für die Verarbeitung dieser zusätzlichen Argumente steht eine Modula-2-Bibliothek zur Verfügung.
- Mittels der gerade genannten Bibliothek lassen sich natürlich auch „Betriebssystem-Programme“ schreiben. Als Beispiel wurde ein Datei-Kopierprogramm gezeigt. Das soll ein Gefühl dafür vermitteln, dass hinter „**Betriebssystem-Kommandos**“ nichts anderes als Programme (z.B. in C, Modula-2, ...) stecken.

Teil X : Hashing

1. **Einführung**
2. **Hashing**

© K. Murmann, H. Neumann, Fakultät für Informatik, Universität Ulm, 2001

1. Einführung

- Prinzip
- Anwendbarkeit und Verwendung

Prinzip

- Hashing (=„zerhacken“) bedeutet, dass Daten(sätze) über einen vorgegebenen Speicherbereich (= Hash-Tabelle) „gestreut“ werden („Streuspeicher-Verfahren“).

- Bis jetzt wurden Daten aufgrund eines Ordnungsprinzips sortiert und in einer (Baum-)Struktur abgelegt, sodass sie mittels des Ordnungsprinzips schnell wieder gefunden werden konnten.
  Beim Hashing wird aus einem Datensatz-Teil (= key/Schlüssel) mittels einer Funktion (= Hash-Funktion) die Stelle des Speicherplatzes ausgerechnet, an dem der Datensatz in der Hash-Tabelle abgelegt wird. Zum Finden des Datensatzes muss lediglich mit der Hash-Funktion die Stelle wieder aus dem key berechnet werden.

Anwendbarkeit und Verwendung

- Hashing ist eine Methode zum sehr schnellen Abspeichern und Wiederfinden von Information (Zeitkomplexität im besten Fall O(1) !!)
- Hashing eignet sich auch für die Speicherung auf externen Medien (Massenspeicher), insbesondere dann, wenn Daten meist nur eingefügt, jedoch nur selten oder nie gelöscht werden.
- Die Anzahl der zu speichernden Elemente muss im Voraus abschätzbar sein. Als Speicher (= Hash-Tabelle) wird ein ARRAY verwendet, das im Voraus richtig dimensioniert werden muss.
  (es gibt als Alternative das „extendable hashing“, bei dem die genannte Bedingung abgemildert ist)
- Hashing wird häufig verwendet im Compilerbau für die Verwaltung der Symboltabelle. Als Key werden die Bezeichner für Typen, Konstanten, Variable, Prozeduren verwendet. Datensätze werden neu eingetragen, vorhandene ergänzt und es muss schnell entscheidbar sein, ob ein Datensatz (bereits) vorhanden ist

2. Hashing

- Hash-Funktion
- Entwurf einer Hash-Funktion
- Kollisionen
- Beispiel

Hash-Funktion

- Gegeben ist:
  - die Key-Menge K von Schlüsseln.
  - die Hash-Tabelle A, ein ARRAY [0 .. m-1] OF <elemtype>, dessen Länge dem Problem genügt, also m > |K|. In A werden die Datensätze abgespeichert

- Aufgabe:
  Definition einer möglichst einfach/schnell zu berechnenden Hash-Funktion

  $$h: \; K \rightarrow i \quad i \in [0 .. m-1] \quad \text{(= Index = Speicherstelle)}$$

  die für einen gegebenen key $k \in K$ eine Adresse (= Index = Speicher-position) bestimmt.
  Die Elemente $k \in K$ sollen möglichst gleichmäßig auf die Index-Menge [0 .. m-1] „gestreut“ werden.

Entwurf einer Hash-Funktion

- Kongruenzmethode:
  m wird als Primzahl gewählt und die Hash-Funktion h definiert als:

  $$h(k) = f(k) \text{ MOD } m$$

  wobei f(k) eine Funktion auf den keys ist. Falls K nicht-numerische Elemente enthält, müssen die Elemente geeignet „umgewandelt“ werden (siehe Beispiel).

- Multiplikationsmethode:
  es wird eine beliebige Konstante $\alpha \in (0,1)$ gewählt, z.B. die Zahl des goldenen Schnitts:

  $$\alpha = \frac{1}{2}(\sqrt{5} - 1) = 0.618033989...$$

  Die Hash-Funktion h wird definiert als:

  $$h(k) = \text{TRUNC}(m \bullet (s(k) - \text{TRUNC}(s(k))) \quad \text{mit} \quad s(k) = \alpha\, f(k)$$

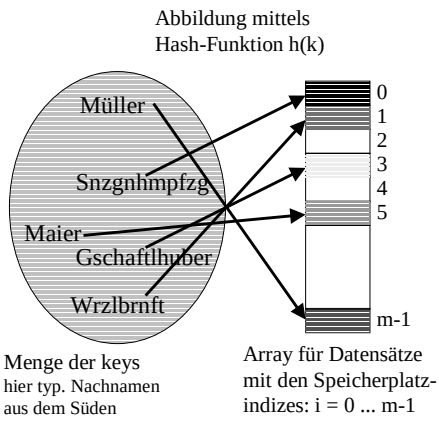
Kollisionen

- Im Falle einer idealen Hash-Funktion h gibt es keine 2 keys mit dem selben Indexwert, die Abbildung ist bijektiv und damit treten keine Kollisionen auf. Im kollisionsfreien Fall benötigt man für die Grundoperationen den zeitlichen Aufwand O(1):
  - Einfügen: A[h(k)] := <einzutragender Datensatz>
  - Suchen: IF A[h(k)] <> empty THEN ... END
  - Löschen: A[h(k)] := empty
- Häufig werden verschiedene keys auf identische Indexpositionen abgebildet, die Abbildung ist dann **nicht** bijektiv und damit treten Kollisionen auf. Leider treten Kollisionen im Durchschnitt sehr häufig auf! Ein Gefühl hierfür bekommt man über das Geburtstags-Paradoxon:

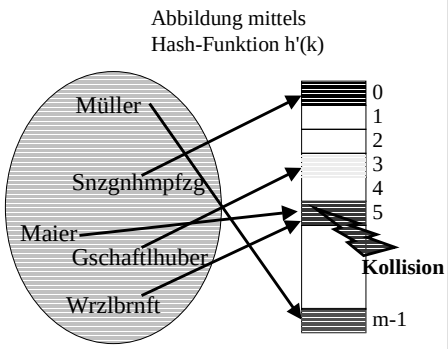
  Wieviele zufällig gewählte Personen müssen zusammenkommen, bis es wahrscheinlicher ist, dass 2 Personen davon am selben Tag Geburtstag haben als dass keine 2 Personen davon am selben Tag Geburtstag haben?
  Antwort: $p > 1/2$ falls die Anzahl der Personen ≥ 23

  übertragen auf Hashing: $m = 365 \rightarrow$ bei **nur** 23 keys: $p(\text{Kollision}) > 1/2$

Hashing ohne Kollisionen



Hashing mit Kollisionen



Beispiel: Hashing mit verschiedenen Hash-Funktionen h(k)

gegeben: key-Menge: K = [Jan, Feb, Mar, ... , Dez] mit |K| = 12
Hash-Tabelle: A[0..16] mit m=17 (Primzahl)

verschiedene Hash-Funktionen h(k):

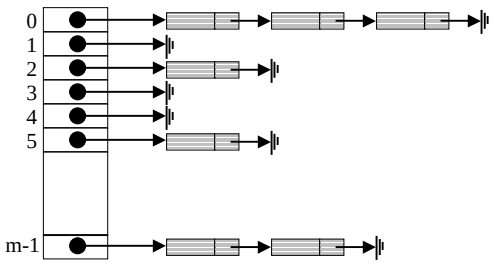
h1(k) = (f(1.Buchst. von k) + f(2.Buchst. von k)) MOD 17
h2(k) = (2·f(1.Buchst. von k) + f(2.Buchst. von k)) MOD 17
h3(k) = (2·f(1.Buchst. von k) + 2· f(2.Buchst. von k) + f(3.Buchst. von k)) MOD 17

Abbildungen:

k	h1(k)	h2(k)	h3(k)
Jan	11	15	2
Feb	11	7	7
Mar	14	6	16
Apr	0	0	1
Mai	14	10	3
Jun	14	1	8
Jul	14	16	6
Aug	5	11	0
Sep	7	4	13
Okt	9	14	4
Nov	12	3	12
Dez	9	14	10
Kollisionen:	mehrere	1	keine

Hashing mit Kollisionslisten

Einfache Methode zur Kollisionsbehandlung:
Datenobjekte mit gleichen Hash-Funktionswerten werden in linearen, verketteten Listen verwaltet:



besetzte Array-Einträge enthalten Zeiger zu linearen, verketteten Listen
leere Array-Einträge enthalten NIL-Zeiger

im Beispiel enthalten
- die Positionen 1, 3, 4 keinen Eintrag
- die Positionen 2, 5 einen Eintrag
- die Position 0 drei Einträge
- die Position m-1 zwei Einträge

- **Hashing** bedeutet die **Abspeicherung von Datensätzen** (z.B. Personaldaten) in einem bereitgestellten, (genügend groß) vordimensionierten Array A[i] (**Hash-Tabelle**). Die Speicherung erfolgt **nicht in einer geordneten Reihenfolge.** Stattdessen wird mittels einer **Hash-Funktion** aus einem bestimmten Element (**key**, Schlüssel) des Datensatzes (z.B. Nachname) ein **Index**wert i berechnet, der die Speicherposition des Datensatzes im Array A[i] festlegt. Die Hash-Funktion dient sowohl der Speicherung als auch dem Suchen von Datensätzen in der Hash-Tabelle.
- Der Entwurf einer geeigneten **Hash-Funktion h** ist von wesentlicher Bedeutung. Einerseits soll h möglichst einfach/schnell im Rechner auswertbar sein, um Datensätze möglichst schnell abspeichern oder finden zu können, andererseits soll sie die Datensätze möglichst gleichmäßig auf die verfügbaren Array-Speicherplätze verteilen, also eine bijektive Abbildung verkörpern und damit keine **Kollisionen** erzeugen. Unter einer Kollision versteht man die Zuweisung einer Speicherposition i an ≥ 2 keys (also Datensätze).
- Kollisionen lassen sich i.d.R. nicht vermeiden. Das kann berücksichtigt werden, indem man in der Hash-Tabelle keine Datensätze speichert sondern lediglich **Zeiger auf lineare, verkettete Listen**, deren Info-Elemente jeweils die Datensätze mit gleichem Index i enthalten.
- Die **Zeit-Komplexität von Hashing** ist von der Ordnung O(1) im kollisionsfreien Fall, sonst größer.