Modellierung und Programmierung

Dr. Martin Riplinger

9.1.2013

Fortgeschrittene Ein- und Ausgabe

Bisher: Ein- und Ausgabe nur über die Kommandozeile

Erweiterung: Konzept des **Datenstroms** (engl. *data stream*)

- Bezeichnet allgemein eine Folge von Datensätzen gleichen Typs, deren Ende nicht im Voraus bekannt ist.
- Lässt sich nicht als Ganzes, sondern nur sequentiell verarbeiten.
- Kann „vor- und zurückgespult“ werden.
- Ein spezieller „End-of-file“-Indikator dient zur Markierung des Endes.
- In C wird auf Datenströme mit Variablen vom Typ `FILE *` zugegriffen. `FILE` ist eine Struktur und enthält u.a. einen **Zeiger auf die aktuell zu verarbeitende Position** im Datenstrom.

Standard-Datenströme `#include <stdio.h>`

`stdin` (**st**andar**d** **in**put): Einlesen von Daten über die Tastatur

`stdout` (**st**andar**d** **out**put): (gepufferte) Ausgabe von Daten auf der Kommandozeile

`stderr` (**st**andar**d** **err**or): Ausgabe auf der Kommandozeile zur Fehlerbehandlung

Dateiströme: Zeiger vom Typ `FILE *` können mit Hilfe spezieller Funktionen mit **Dateien** verbunden werden.

Fortgeschrittene Ein- und Ausgabe: Funktionen

Öffnen einer Datei

```
FILE *fopen(const char *path, const char *mode);
```

- Erzeugt einen Datenstrom aus der Datei, die unter dem Pfad `path` liegt.
- Rückgabewert ist ein `FILE *`, der entweder an den **Anfang** oder das **Ende des neuen Dateistroms** zeigt. Bei Scheitern - z. B. Öffnen einer nicht vorhandenen Datei oder fehlende Dateirechte - liefert die Funktion `NULL` zurück.
- Der String `path` gibt den Pfad zur Datei im Dateisystem an, z. B. `"../Daten/werte.dat"` (**relativer Pfad**) oder `"/usr/include/stdio.h"` (**absoluter Pfad**).
- Der Modus `mode` gibt an, in welche Richtung der Strom fließen kann. Mögliche Modi sind `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"` und `"a+"`:

Modus	Bedeutung	Zeigerposition	Dateiinhalt	Datei existiert nicht
`"r"`	**r**ead only	Dateianfang	unverändert	Rückgabewert `NULL`
`"w"`	**w**rite only	Dateianfang	wird gelöscht	Neue Datei
`"a"`	**a**ppend only	Dateiende	unverändert	Rückgabewert `NULL`
`"r+"`	read/write	Dateianfang	unverändert*	Rückgabewert `NULL`
`"w+"`	read/write	Dateianfang	wird gelöscht	Neue Datei
`"a+"`	append/write	Dateiende	unverändert	Neue Datei

*Bezieht sich nur auf den Zeitpunkt des Öffnens

- **Beispiel**: `FILE *strom = fopen("Dokument.txt", "r");`

Fortgeschrittene Ein- und Ausgabe: Funktionen

Schließen eines Datenstroms

```
int *fclose(FILE *stream);
```

Schließt den Datenstrom `stream` und gibt `0` zurück bei Erfolg, andernfalls die Konstante `EOF` (**E**nd **O**f **F**ile).
Wichtig: Zu **jedem** `fopen()` gehört ein `fclose()`!

Position herausfinden

```
long ftell(FILE *stream);
```

Gibt die aktuelle Position des Zeigers in `stream` als **Adressabstand vom Anfang des Stromes** (offset) **in Byte** an.

Vor- und Zurückspulen

```
int fseek(FILE *stream, long offset, int whence);
```

Versetzt die Position des Zeigers in `stream` um `offset` Byte. Je nach Wert von `whence` wird der Versatz relativ zum **Anfang** (`whence = SEEK_SET`), zum **Ende** (`whence = SEEK_END`) oder zur **aktuellen Position** (`whence = SEEK_CUR`) gerechnet. Der Rückgabewert ist `0` bei Erfolg, andernfalls ungleich `0`.

```
void rewind(FILE *stream);
```

Spult den Strom `stream` an den Anfang zurück. Der Aufruf `rewind(stream);` ist äquivalent zu `fseek(stream, 0L, SEEK_SET);` (bis auf interne Feinheiten).

Fortgeschrittene Ein- und Ausgabe: Funktionen

Aus einem Datenstrom lesen (Text)

```
int fscanf(FILE *stream, const char *format, ...);
```

Genau wie `scanf`, wobei statt aus `stdin` aus dem Datenstrom `stream` gelesen wird. (Deshalb ist `scanf(...)` äquivalent zu `fscanf(stdin, ...)`.) Rückgabewert ist die Anzahl der gelesenen Zeichen.

```
int fgetc(FILE *stream);
```

Liest ein Zeichen aus `stream`, das als `int` gecastet zurückgegeben wird. Im Fehlerfall oder bei Dateiende ist der Rückgabewert `EOF`.

```
char *fgets(char *s, int n, FILE *stream);
```

Liest aus `stream` und schreibt das Ergebnis in den Puffer, auf den `s` zeigt. Das Lesen wird abgebrochen, wenn entweder das Zeilenende `\n` oder das Dateiende erreicht ist oder $n-1$ Zeichen gelesen wurden. Im Fehlerfall oder bei Dateiende ist der Rückgabewert `NULL`. **Beachte**: Nach dem Aufruf einer dieser Funktionen zeigt `stream` auf die **erste Position im Anschluss an den bearbeiteten Bereich**.

Fortgeschrittene Ein- und Ausgabe: Funktionen

In einen Datenstrom schreiben (Text)

```
int fprintf(FILE *stream, const char *format, ...);
```

Arbeitet wie `printf`, wobei in den Datenstrom `stream` geschrieben wird. (Deshalb ist `printf(...)` äquivalent zu `fprintf(stdout, ...)`.) Rückgabewert ist die Anzahl der geschriebenen Zeichen.

```
int fputc(int c, FILE *stream);
```

Schreibt das als `unsigned char` gecastete Zeichen `c` in den Strom `stream`. Das Zeichen wird als `int` zurückgegeben, im Fehlerfall `EOF`.

```
int *fputs(const char *s, FILE *stream);
```

Schreibt den String, auf den `s` zeigt, ohne das Nullzeichen `\0` in den Strom `stream`. Im Fehlerfall wird `EOF` zurückgegeben.

Beachte: Nach dem Aufruf einer dieser Funktionen zeigt `stream` auf die **erste Position im Anschluss an den bearbeiteten Bereich**.

Fortgeschrittene Ein- und Ausgabe: Beispiel

```c
#include <stdio.h>

int main(void)
{
  FILE *datenstrom;

  datenstrom = fopen("Testdatei.txt", "r");
  if (datenstrom == NULL)
  {
    fprintf(stderr, "Fehler: Testdatei.txt konnte nicht geoeffnet werden!\n");
    return -1;
  }

  fseek(datenstrom, 0, SEEK_END); // Ans Ende spulen
  fprintf(stdout, "Offset am Ende: %lu\n", ftell(datenstrom));

  fseek(datenstrom, -5, SEEK_CUR); // 5 Bytes zurueck
  fprintf(stdout, "     nach Zurueckspulen: %lu\n", ftell(datenstrom));

  rewind(datenstrom);             // Zurueckspulen
  fprintf(stdout, "     am Anfang: %lu\n", ftell(datenstrom));

  fclose(datenstrom);

  return 0;
}
```

Fortgeschrittene Ein- und Ausgabe: Beispiel

Datei *Testdatei.txt*: `Hallo Welt!`

Die Datei enthält genau eine Zeile „Hallo Welt!“ (11 Characters = 11 Byte).

Ausgabe des Programms:

```
Offset am Ende: 11
       nach Zurueckspulen: 6
       am Anfang: 0
```

Beispiel: Matrix aus Datei einlesen (Funktion)

```
1  double **neueMatrixAusDatei(FILE *strom, int *n_zeilen, int *n_spalten)
2  {
3    int i, j;
4    double **matrix;
5
6    while (fgetc(strom) == '#') // Kommentarzeichen->Zeile ueberspringen
7      while (fgetc(strom) != '\n');
8    fseek(strom, -1, SEEK_CUR); // Eine Position zurueck
9
10   fscanf(strom, " n = %d, m = %d\n", n_zeilen, n_spalten);
11   matrix = neueMatrix(*n_zeilen, *n_spalten);
12
13   for (i=0; i<*n_zeilen; i++)
14   {
15     for (j=0; j<*n_spalten; j++)
16       fscanf(strom, "%lf", &matrix[i][j]);
17   }
18   return matrix;
19 }
```

- Jeder Aufruf von `fgetc` erhöht die Position von `strom` um 1. Deshalb zeigt `strom` nach der (leeren) Schleife in Zeile 7 auf den **Anfang der nächsten Zeile**. Insgesamt überspringen die Zeilen 6 und 7 Dateizeilen, die mit '#' beginnen.
- Zeile 10 erwartet einen Text wie `n = 6, m = 1`. Die gelesenen Zahlen werden in `n_zeilen` und `n_spalten` (beides Zeiger auf `int`-Variablen) gespeichert.
- Für `fscanf` gilt: Leerzeichen und Zeilenumbrüche **innerhalb der Platzhalter** werden automatisch übersprungen. Ein Leerzeichen im Formatstring steht für **eine beliebige Anzahl** (auch 0) von tatsächlich zu lesenden Leerzeichen.

Beispiel: Matrix aus Datei einlesen (Hauptprogramm)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void zeigeMatrix(double **matrix, int n_zeilen, int n_spalten);
5  void loescheMatrix(double **matrix, int n_zeilen);
6  double **neueMatrixAusDatei(FILE *strom, int *n_zeilen, int *n_spalten);
7
8  int main(void)
9  { int n, m;
10   FILE *fp;
11   if ( (fp = fopen("Matrix.dat", "r")) == NULL )
12     return 1;
13   double **H = neueMatrixAusDatei(fp, &n, &m);
14
15   zeigeMatrix(H, n, m);
16   loescheMatrix(H, n);
17   fclose(fp);
18   return 0;
19 }
20
21 // Funktionsdefinitionen ...
```

Ausgabe:

```
1.0000 0.5000 0.3333 0.2500 0.2000 0.1667 0.1429
0.5000 0.3333 0.2500 0.2000 0.1667 0.1429 0.1250
0.3333 0.2500 0.2000 0.1667 0.1429 0.1250 0.1111
0.2500 0.2000 0.1667 0.1429 0.1250 0.1111 0.1000
```

Beispiel: Matrix aus Datei einlesen (`Matrix.dat`)

Version 1

```
n=4, m=7
1.0      0.500000 0.333333 0.250000 0.2      0.166667 0.142857
0.5      0.333333 0.25     0.2      0.166667 0.142857 0.125
0.333333 0.25     0.2      0.166667 0.142857 0.125    0.111111
0.25     0.2      0.166667 0.142857 0.125    0.111111 0.1
```

Version 2

```
# Das ist die Hilbertmatrix H_ij = 1/(i+j-1)
#
# Hoffentlich werden diese Zeilen uebersprungen...
n = 4, m = 7

1.0      0.500000 0.333333 0.250000 0.2      0.166667 0.142857
0.5      0.333333 0.25     0.2      0.166667 0.142857 0.125
0.333333 0.25     0.2      0.166667 0.142857 0.125    0.111111
0.25     0.2      0.166667 0.142857 0.125    0.111111 0.1
```

$\rightsquigarrow$ Beide Versionen werden korrekt eingelesen!

Beispiel: Matrix in Datei schreiben (Funktion)

```
1  void schreibeMatrixInDatei(FILE *strom, double **matrix,
2                             int n_zeilen, int n_spalten)
3  {
4    int i, j;
5    fprintf(strom, "# Automatisch generiert mit schreibeMatrixInDatei\n\n");
6    fprintf(strom, "n = %d, m = %d\n\n", n_zeilen, n_spalten);
7
8    for (i=0; i<n_zeilen; i++)
9    {
10     for (j=0; j<n_spalten; j++)
11       fprintf(strom, "%.6lf ", matrix[i][j]);
12
13     fprintf(strom, "\n");
14   }
15
16   return;
17 }
```

Beispiel: Matrix in Datei schreiben (Hauptprogramm)

```c
#include <stdio.h>
#include <stdlib.h>

void loescheMatrix(double **matrix, int n_zeilen);
double **neueMatrixAusDatei(FILE *strom, int *n_zeilen, int *n_spalten);
void schreibeMatrixInDatei(FILE *strom, double **matrix,
                           int n_zeilen, int n_spalten);

int main(void)
{
  int n, m;
  FILE *fp;
  double **H;

  fp = fopen("Matrix.dat", "r");
  H  = neueMatrixAusDatei(fp, &n, &m);
  fclose(fp);

  fp = fopen("Matrix_generiert.dat", "w");
  schreibeMatrixInDatei(fp, H, n, m);
  fclose(fp);

  loescheMatrix(H, n);

  return 0;
}
```

Beispiel: Matrix in Datei schreiben

Inhalt von `Matrix_generiert.dat`:

```
# Automatisch generiert mit schreibeMatrixInDatei

n = 4, m = 7

1.000000 0.500000 0.333333 0.250000 0.200000 0.166667 0.142857
0.500000 0.333333 0.250000 0.200000 0.166667 0.142857 0.125000
0.333333 0.250000 0.200000 0.166667 0.142857 0.125000 0.111111
0.250000 0.200000 0.166667 0.142857 0.125000 0.111111 0.100000
```

Speicherung als Text oder binär: Vergleich

Dateigröße

Binär: Produkt aus Anzahl der Elemente und Größe eines Elements **in Byte**.

Text: Variiert mit der **textuellen** Länge der gespeicherten Elemente. Meistens größer als im Binärformat.

Präzision

Binär: Entspricht der Genauigkeit des Datentyps.

Text: Der Genauigkeit sind prinzipiell keine Grenzen gesetzt, jedoch kann der Speicherbedarf dadurch sehr groß werden.
Beispiel: Um die größtmögliche `float`-Zahl ($3.4 \cdot 10^{38}$) als Text zu speichern, benötigt man 38 Zeichen $\hat{=}$ 38 Byte, im Binärformat nur `sizeof(float)` $=$ 4 Byte!

Metadaten

Binär: Üblicherweise ist die Datei unterteilt in einen **Header** und die **Daten**. Es muss genau beschrieben werden, welche Bytes wofür stehen.
Beispiel Matrix: Bytes 1-4 – `unsigned` – Länge des Headers in Byte, 5-8 – `unsigned` – Anzahl der Zeilen, 9-12 – `unsigned` – Anzahl der Spalten, 13-Ende – `double` – Matrixelemente.

Text: Informationen über die Daten können in die Datei geschrieben werden.

Vor- und Nachteile der binären Speicherung

Vorteile:

- Allein der Datentyp bestimmt die Größe des benötigten Speicherplatzes.
- Es wird meistens deutlich weniger Speicherplatz als für die Textvariante benötigt.
- Maschinennahe Ein- und Ausgabe, daher deutlich schneller.

Nachteile:

- Binäre Daten sind nicht direkt vom Menschen lesbar.
- Binärformate sind vom System abhängig, d.h. Daten müssen unter Umständen zuerst umgewandelt werden.

Aus einem Datenstrom lesen (binär)

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Liest `nmemb` Elemente der Größe `size` aus dem Datenstrom `stream` und speichert sie (in der gleichen Reihenfolge) im Feld, auf das `ptr` zeigt. Rückgabewert ist die Anzahl der erfolgreich gelesenen Elemente.

Beachte: Die Daten werden nicht als Text, sondern als **Bitfolgen** interpretiert.

In einen Datenstrom schreiben (binär)

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Schreibt `nmemb` Elemente der Größe `size` aus dem Feld, auf das `ptr` zeigt, in den Datenstrom `stream`. Rückgabewert ist die Anzahl der erfolgreich geschriebenen Elemente.

Beispiel: binäres Lesen und Schreiben (Funktionen)

```
1  void schreibeMatrixInBinDatei(char *pfad, double **matrix,
2                                int n_zeilen, int n_spalten)
3  { int i;
4    FILE *fp;
5
6    if ( (fp = fopen(pfad, "wb")) == NULL ) return;
7
8    for (i=0; i<n_zeilen; i++)
9      fwrite(matrix[i], sizeof(double), n_spalten, fp);
10
11   fclose(fp);
12   return;
13 }
14
15 double **neueMatrixAusBinDatei(char *pfad, int n_zeilen, int n_spalten)
16 { int i;
17   double **matrix;
18   FILE *fp;
19
20   if ( (fp = fopen(pfad, "rb")) == NULL ) return NULL;
21
22   matrix = neueMatrix(n_zeilen, n_spalten);
23   for (i=0; i<n_zeilen; i++)
24     fread(matrix[i], sizeof(double), n_spalten, fp);
25
26   fclose(fp);
27   return matrix;
28 }
```

Beispiel: binäres Lesen und Schreiben (Hauptprogramm)

```c
double **neueMatrixAusBinDatei(char *pfad, int n_zeilen, int n_spalten);
void schreibeMatrixInBinDatei(char *pfad, double **matrix,
                              int n_zeilen, int n_spalten);

int main(void)
{
  int n = 4, m = 7;
  double **H;

  H  = neueMatrixAusBinDatei("Matrix.bin", n, m);
  zeigeMatrix(H, n, m);
  schreibeMatrixInBinDatei("Matrix_kopie.bin", H, n, m);
  loescheMatrix(H, n);

  return 0;
}
```

Beachte:

- Die Funktionen werden (zur Abwechslung) nicht mit dem bereits geöffneten Strom, sondern mit dem Pfad zum Dateinamen aufgerufen.
- In den `if`-Statements wird zunächst der Datenstrom geöffnet und anschließend auf `NULL` überprüft.
- Da jede Matrixzeile als separates Feld gespeichert ist (zweistufige Speicherung mit `double **`), muss auch zeilenweise in die Datei geschrieben werden.

C-Präprozessor

- Wir haben bereits die Direktive `#include <Headerdatei>` verwendet, um „fremde“ Funktionsdeklarationen zu importieren
- Generell ist der Präprozessor dafür zuständig, **Text durch anderen Text zu ersetzen**.
- Der Präprozessor wird **vor dem Compiler** aufgerufen, deshalb muss der verarbeitete Code syntaktisch korrekt sein.
- Präprozessordirektiven beginnen stets mit einer Raute `#` und stehen im Code bis auf wenige Ausnahmefälle **ganz zu Beginn**.
- Die wichtigsten Direktiven sind **Einfügungen**, **Makros** und **bedingte Ersetzungen**.

C-Präprozessor: Einfügungen

Standard-Header

- **Syntax**: `#include <systemheader.h>`
- Fügt den Text der Headerdatei `systemheader.h` an der Stelle im Code ein, an welcher der `#include`-Befehl steht.
- Die Datei wird im Standard-Header-Pfad des Systems gesucht. Bei Unix-Systemen ist dies üblicherweise `/usr/include`.

  Beispiel: `#include <stdio.h>` bindet die Datei `/usr/include/stdio.h` ein.

  Bei Headern in Unterordnern muss dieser mit angegeben werden, z. B. `#include <sys/time.h>` für den Header `/usr/include/sys/time.h`.

Lokale (eigene) Header

- **Syntax**: `#include "lokaler_header.h"`
- Bindet die Datei `lokaler_header.h` ein, die im **aktuellen Verzeichnis** liegt. Will man einen Header aus einem anderen Pfad inkludieren, muss dem Compiler der Pfad mitgeteilt werden.

  Beispiel: Zum Einbinden des Headers `matrix.h` im Unterordner `meine_header` wird
  - im Quellcode die Zeile `#include "matrix.h"` eingefügt.
  - der Compiler mit der Option `-Imeine_header` aufgerufen.

  Die Option `-IOrdner` macht die Header in `Ordner` für den Compiler sichtbar.

C-Präprozessor: Einfügungen – Beispiel `matrix.h`

Headerdatei `matrix.h` im Unterordner `meine_header` (Ausschnitt):

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Legt Speicher fuer eine neue Matrix an und liefert einen Zeiger auf die
5  // erste Zeile; bei Misserfolg NULL
6  double **neueMatrix(int n_zeilen, int n_spalten);
7
8  // Gibt die Matrix am Bildschirm aus
9  void zeigeMatrix(double **matrix, int n_zeilen, int n_spalten);
```

Programmcode `matrix_test.c`:

```
1  #include "matrix.h"
2
3  int main(void)
4  {
5    int n = 4, m = 7;
6    double **H = neueMatrixAusBinDatei("Matrix.bin", n, m);
7    zeigeMatrix(H, n, m);
8    loescheMatrix(H, n);
9
10   return 0;
11 } // Hiernach die Funktionsdefinitionen!!
```

Compiler-Aufruf: `gcc -o matrix_test matrix_test.c -Imeine_header`