Einführung in die Programmierung

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2009/10

Übersicht

- Arrays
- Datentypen, Operatoren und Kontrollstrukturen
- Funktionen und Zeiger
- Zeichenketten und Strukturen
- *die Standardbibliothek*
- strukturierte Programmierung
- modulare Programmierung

Die Standardbibliothek

gutes Beispiel

- für prozedurale Programmierung
- und wiederverwendbarer Software

Definitionsdateien können in beliebiger Reihenfolge mittels `#include` eingefügt werden.

Eine Definitionsdatei muss eingefügt sein, bevor irgendetwas benutzt wird, das in der Header-Datei vereinbart wird.

Übersicht

- *Dateioperationen*
- Hilfsfunktionen
- Zeichenketten
- Zufallszahlen

Ein- und Ausgabe

ein *Datenstrom (stream)*

- ist Quelle oder Ziel von Daten und
- wird mit einem Peripheriegerät verknüpft

zwei Arten von Datenströmen werden unterschieden:

- *für Text:* eine Folge von Zeilen, jede Zeile enthält beliebig viele Zeichen und ist mit `\n` abgeschlossen
- *für binäre Informationen:* eine Folge von Bytes zur Darstellung interner Daten

Ein- und Ausgabe

Ein Datenstrom wird durch

- *Öffnen (open)* mit der Datei/dem Gerät verbunden,
- *Schließen (close)* von der Datei/dem Gerät getrennt.

Öffnet man eine Datei, erhält man einen Zeiger auf ein Objekt von Typ `FILE`.

- Das `FILE`-Objekt enthält alle Informationen, die zur Kontrolle des Datenstroms notwendig sind.
- Beim Programmstart sind die Datenströme `stdin`, `stdout` und `stderr` bereits geöffnet.

Dateioperationen

```
FILE *fopen(const char *filename, const char *mode)
```

`fopen` öffnet die angegebene Datei und liefert einen Datenstrom oder `NULL` bei Misserfolg.

Zu den erlaubten Werten von `mode` gehören:

- `r` zum Lesen öffnen (read)
- `w` zum Schreiben öffnen, alten Inhalt wegwerfen (write)
- `a` zum Anfügen öffnen bzw. erzeugen (append)

Wird an die Zeichen ein `b` angehängt, dann wird auf eine binäre Datei zugegriffen, sonst auf eine Textdatei.

Dateioperationen

```
int fflush(FILE *stream)
```

- gepufferte, aber noch nicht geschriebene Daten werden geschrieben
- liefert `EOF` bei einem Schreibfehler, sonst 0

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

- schreibt noch nicht geschriebene Daten
- wirft noch nicht gelesene, gepufferte Daten weg
- gibt automatisch angelegte Puffer frei
- schließt den Datenstrom
- liefert `EOF` bei Fehlern, sonst 0

Dateioperationen

`int remove(const char *filename)`

- entfernt die angegebene Datei
- liefert bei Fehlern einen Wert ungleich 0

`int rename(const char *oldname, const char *newname)`

- ändert den Namen einer Datei
- liefert im Fehlerfall einen Wert ungleich 0

`FILE *tmpfile(void)`

- erzeugt eine temporäre Datei (automatisches Löschen bei `fclose` oder normalem Programmende)
- liefert einen Datenstrom oder `NULL` im Fehlerfall

Formatierte Ausgabe

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

- wandelt die Ausgaben entsprechend `format` um
- schreibt in den angegebenen Datenstrom
- Resultat: Anzahl der geschriebenen Zeichen, negativer Wert im Fehlerfall
- Typangaben:
  - `c` char
  - `d` int
  - `f` float
  - `s` Zeichenkette (String)
  - `p` Pointer
- analog zu `printf`

Formatierte Eingabe

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

- liest vom Datenstrom `file` unter Kontrolle von `format`
- legt die umgewandelten Werte in den Argumenten ab
- alle Argumente müssen Zeiger sein
- ist beendet, wenn `format` abgearbeitet ist oder eine Umwandlung nicht durchgeführt werden kann
- liefert `EOF`, wenn vor der ersten Umwandlung das Dateiende erreicht wird oder ein Fehler passiert, ansonsten die Anzahl umgewandelter und abgelegter Eingaben
- analog zu `scanf`
- liest über Zeilengrenzen hinweg, um seine Eingabe zu finden

Fehlerbehandlung

Viele der IO-Bibliotheksfunktionen notieren, ob ein Dateiende gefunden wurde oder ein Fehler aufgetreten ist:

- `void clearerr(FILE *file)` löscht die Notizen über Dateiende und Fehler für den Datenstrom `file`.
- `int feof(FILE *file)` liefert einen Wert ungleich 0, wenn für `file` ein Dateiende notiert ist.
- `int ferror(FILE *file)` liefert einen Wert ungleich 0, wenn für `file` ein Fehler notiert ist.

In `errno.h` ist eine global gültige Variable `errno` definiert. Sie enthält eine Fehlernummer, die Informationen über den zuletzt aufgetretenen Fehler zulässt.

Fehlerbehandlung

`void perror(const char *s)` gibt `s` und eine von der Implementierung definierte Fehlermeldung aus, die sich auf den Wert in `errno` bezieht.

```
#include <stdio.h>

void main(void) {
    FILE *f;

    f = fopen("test.txt", "r");
    if (f == NULL)
        perror("test.txt");
    else fclose(f);
}
```

Wenn die angegebene Datei nicht gefunden wird, erhalten wir hier die Ausgabe: `test.txt: No such file or directory`

Beispiel

Ziel: Eine Liste von Studenten in einer Datei abspeichern und aus einer Datei lesen.

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

typedef struct {
    int tag, monat, jahr;
} date_t;

typedef struct {
    char name[30], vname[30];
    date_t gebdat;
    short fb;
    long matnr;
} stud_t;
```


Beispiel

```
int saveToFile(stud_t *s, int n, char *fname) {
  int i;
  FILE *f;

  if ((f = fopen(fname, "w")) == NULL) {
    perror(fname);
    return 1;
  }

  fprintf(f, "%d\n", n); // Header: Anzahl Daten
  for (i = 0; i < n; i++)
    fprintf(f,"%s %s %02d.%02d.%4d,%hd,%08ld\n",
      s[i].name, s[i].vname, s[i].gebdat.tag,
      s[i].gebdat.monat, s[i].gebdat.jahr,
      s[i].fb, s[i].matnr);

  return (fclose(f) == EOF ? 2 : 0);
}
```


Beispiel

```
stud_t *readFromFile(char *fname, int *len) {
   stud_t *s;
   FILE *f;
   int i;

   f = fopen(fname, "r");
   if (f == NULL) {
      perror(fname);
      return NULL;
   }

   /* Anzahl Daten aus Header lesen */
   fscanf(f, "%d", len);

   /* dynamisches Array anlegen */
   s = (stud_t *) malloc(*len * sizeof(stud_t));
```

Beispiel

```
    // einlesen der Daten (ohne Fehlerbehandlung)
    for (i = 0; i < *len && !error; i++)
       fscanf(f, "%s%s%d.%d.%d,%hd,%ld",
              s[i].name,
              s[i].vname,
              &s[i].gebdat.tag,
              &s[i].gebdat.monat,
              &s[i].gebdat.jahr,
              &s[i].fb,
              &s[i].matnr);

    fclose(f);

    return s;
}   /* End: readFromFile() */
```


Beispiel

```
int main(void) {
   int i, len;
   stud_t *t;
   stud_t s[3] = {
      {"Huber", "Hans", {23, 5, 1984},
            3, 12345678},
      {"Mayer", "Gabi", {12, 9, 1982},
            4, 7654321},
      {"Meier", "Rosi", {17, 2, 1983},
            4, 98761234}
   };

   saveToFile(s, 3, "_student.txt");
```


Beispiel

```
    t = readFromFile("_student.txt", &len);

    for (i = 0; i < len; i++)
       printf("%s %s %02d.%02d.%4d,%hd,%08ld\n",
          t[i].name, t[i].vname, t[i].gebdat.tag,
          t[i].gebdat.monat, t[i].gebdat.jahr,
          t[i].fb, t[i].matnr);

    free(t);
    return 0;
}
```

Problem: es wird jeweils nur der erste Vorname geliefert, da das Einlesen bei einem Leerzeichen abgebrochen wird.

Lösung??? → siehe `strtok()`

Ein- und Ausgabe von Zeichen

`int fgetc(FILE *file)`

- liefert das nächste Zeichen des Datenstroms als `unsigned char` (umgewandelt in `int`)
- im Fehlerfall oder bei Dateiende `EOF`

`char *fgets(char *s, int n, FILE *file)`

- liest höchstens die nächsten `n-1` Zeichen in `s` ein
- hört vorher auf, wenn ein Zeilentrenner gefunden wird
- der Zeilentrenner wird im Vektor abgelegt
- der Vektor wird mit `\0` abgeschlossen
- liefert `NULL` bei Dateiende oder im Fehlerfall, ansonsten `s`

Ein- und Ausgabe von Zeichen

`int fputc(int c, FILE *file)`

- schreibt das Zeichen `c` (umgewandelt in `unsigned char`) in den Datenstrom
- liefert das ausgegebene Zeichen, im Fehlerfall `EOF`

`int fputs(const char *s, FILE *file)`

- schreibt die Zeichenkette `s` in den Datenstrom
- liefert einen nicht-negativen Wert, im Fehlerfall `EOF`

Beispiel

Häufigkeit der Buchstaben in einem Text bestimmen.

```
#include <stdio.h>

int getIndex(char c) {
    /* aus Klein- mach Grossbuchstabe */
    if (c >= 'a' && c <= 'z')
        c -= 'a' - 'A';

    /* alle anderen Zeichen ausschliessen */
    if (c < 'A' || c > 'Z')
        return -1;

    /* A..Z auf 0..25 abbilden */
    return c - 'A';
}
```


Beispiel

```
int main(int argc, char *argv[]) {
    int i, H[26] = {0};
    char c, *filename = "_test.txt";
    FILE *file;

    if (argc > 1)
        filename = argv[1];
    printf("untersuche Datei %s!\n", filename);

    if (!(file = fopen(filename, "r"))) {
        perror(filename);
        return 1;
    }
    ...
```


Beispiel

```
    while ((c = fgetc(file)) != EOF) {
        if ((i = getIndex(c)) >= 0)
            H[i] += 1;
    }

    fclose(file);

    for (i = 0, c = 'a'; c <= 'z'; i++, c++)
        printf("H[%c] = %d\n", c, H[i]);

    return 0;
}
```

Beispiel

anwenden des obigen Programms auf

```
Dies ist nur ein kleiner Test zur Ermittlung von
Häufigkeiten der einzelnen Buchstaben. Probleme
machen eventuell deutsche Umlaute, denn die werden
nicht gezählt.
```

liefert

```
H[a] = 3   H[g] = 3   H[m] = 4   H[s] = 5   H[y] = 0
H[b] = 3   H[h] = 6   H[n] = 16  H[t] = 12  H[z] = 3
H[c] = 4   H[i] = 10  H[o] = 2   H[u] = 9
H[d] = 6   H[j] = 0   H[p] = 1   H[v] = 2
H[e] = 27  H[k] = 2   H[q] = 0   H[w] = 1
H[f] = 1   H[l] = 8   H[r] = 7   H[x] = 0
```


Direkte Ein- und Ausgabe

```
size_t fread(void *ptr, size_t size, size_t nobj,
             FILE *file)
```

- liest aus dem Datenstrom in den Vektor `ptr` höchstens `nobj` Objekte der Größe `size` ein
- liefert die Anzahl der eingelesenen Objekte
- der Zustand des Datenstroms kann mit `feof` und `ferror` untersucht werden

```
size_t fwrite(const void *ptr, size_t size,
              size_t nobj, FILE *file)
```

- schreibt `nobj` Objekte der Größe `size` aus dem Vektor `ptr` in den Datenstrom
- liefert die Anzahl der ausgegebenen Objekte

Direkte Ein- und Ausgabe

Vorteile von Textdateien:

- mit Editor lesbar und änderbar
- plattformunabhängig

Nachteile von Textdateien:

- hoher Speicherplatzbedarf
- hohe Zugriffszeiten
- nur auf `char` und `char *` kann direkt zugegriffen werden, alle anderen Datentypen müssen konvertiert werden

Achtung: Bei `fwrite` werden nur die Werte von Zeigern gespeichert, nicht der Inhalt, auf den der Zeiger zeigt!

Beispiel

```
int saveToFile(stud_t *s, int n, char *fname) {
    int x;
    FILE *file;

    file = fopen(fname, "wb");
    if (file == NULL)
        return -1;

    x = fwrite(s, sizeof(stud_t), n, file);
    if (x != n)
        perror(fname);

    return fclose(file);
}
```


Beispiel

```
int readFromFile(stud_t *s, int n,
                 char *fname) {
    int x;
    FILE *file;

    file = fopen(fname, "rb");
    if (file == NULL)
        return -1;

    x = fread(s, sizeof(stud_t), n, file);
    if (x != n)
        perror(fname);

    return fclose(file);
}
```


Beispiel

```
int main(void) {
   stud_t stud[5], *s;

   ...
   saveToFile(stud, 5, filename);

   s = (stud_t *) malloc(5 * sizeof(stud_t));
   readFromFile(s, 5, filename);
   for (i = 0; i < 5; i++)
      printf("%s %s %02d.%02d.%4d,%hd,%ld\n",
         s[i].name, s[i].vname, s[i].gebdat.tag,
         s[i].gebdat.monat, s[i].gebdat.jahr,
         s[i].fb, s[i].matrikelnr);

   return 0;
}
```


Positionieren in Dateien

```
int fseek(FILE *file, long offset, int origin)
```

- Dateiposition für `file` setzen, nachfolgende Lese- oder Schreiboperation greift auf Daten ab dieser Position zu.
- neue Position ergibt sich aus Addition von `offset` Bytes zu `origin`
- mögliche Werte für `origin`: Dateianfang `SEEK_SET`, aktuelle Position `SEEK_CUR`, Dateiende `SEEK_END`
- liefert einen von 0 verschiedenen Wert bei Fehler

```
long ftell(FILE *file)
```

- liefert die aktuelle Dateiposition oder `-1L` bei Fehler

Positionieren in Dateien

```
void rewind(FILE *file)
```

- analog zu `fseek(file, 0L, SEEK_SET);` und anschließendem `clearerr(file);`

```
int fgetpos(FILE *file, fpos_t *ptr)
```

- speichert aktuelle Position für den Datenstrom bei `*ptr`
- liefert einen von 0 verschiedenen Wert bei Fehler

```
int fsetpos(FILE *file, const fpos_t *ptr)
```

- positioniert `file` auf die Position, die von `fgetpos` in `*ptr` abgelegt wurde
- liefert einen von 0 verschiedenen Wert bei Fehler

Beispiel

```c
#include <stdio.h>

int main(void) {
    int z;
    char line[10];      // feste Satzlaenge: 10
    FILE *file;         // keine Fehlerbehandlung

    file = fopen("_seek.txt", "r");

    printf("gehe zu Zeile ");
    scanf("%d", &z);
    fseek(file, (z-1) * 10, SEEK_SET);
    fgets(line, 10, file);
    printf("%s\n", line);
    fclose(file);

    return 0;
}
```


Übersicht

- Dateioperationen
- *Hilfsfunktionen*
- Zeichenketten
- Zufallszahlen

Tests für Zeichenklassen: `ctype.h`

Funktionen zum Testen von Zeichen. Jede Funktion

- hat ein `int`-Argument, dessen Wert entweder `EOF` ist oder als `unsigned char` dargestellt werden kann.
- hat den Rückgabetyp `int`.
- liefert einen Wert ungleich 0, wenn das Argument die beschriebene Bedingung erfüllt.

Zusätzlich: Funktionen zur Umwandlung zwischen Groß- und Kleinbuchstaben.

`tolower(c)`	Umwandlung in Kleinbuchstaben
`toupper(c)`	Umwandlung in Großbuchstaben

Tests für Zeichenklassen

Funktion	Beschreibung
`isalnum(c)`	`isalpha(c)` oder `isdigit(c)` ist erfüllt
`isalpha(c)`	`isupper(c)` oder `islower(c)` ist erfüllt
`iscntrl(c)`	Steuerzeichen
`isdigit(c)`	dezimale Ziffer
`isgraph(c)`	sichtbares Zeichen, kein Leerzeichen
`islower(c)`	Kleinbuchstabe, kein Umlaut oder ß
`isprint(c)`	sichtbares Zeichen, auch Leerzeichen
`isspace(c)`	Leerzeichen, Seitenvorschub, ...
`isupper(c)`	Großbuchstabe, kein Umlaut oder ß
`isxdigit(c)`	hexadezimale Ziffer

Zeichenketten umwandeln in Zahlen: `stdlib.h`

- `double strtod(const char *str, char **endp)` und
- `long strtol(const char *str, char **endp, int base)`

wandeln den Anfang der Zeichenkette `str` in `double`/`long` um, dabei wird Zwischenraum am Anfang ignoriert.

- `strtoul` analog zu `strtol`, Resultattyp `unsigned long`

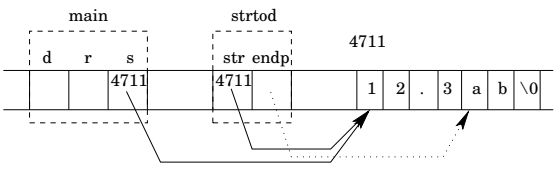
Die Funktionen speichern einen Zeiger auf den nicht umgewandelten Rest der Zeichenkette bei `*endp`.

Wert von `base`: $2, \ldots, 36$. Umwandlung erfolgt unter der Annahme, das die Eingabe in dieser Basis repräsentiert ist.

Erinnerung: call by reference

Warum ist `endp` als Zeiger auf Zeiger auf `char` definiert?

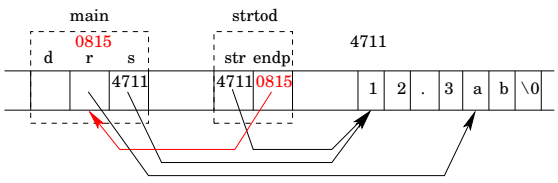
```c
double d;
char *r = NULL;
char *s = "12.3abc";

d = strtod(s, r);   // falsch!!!!!!
```



Erinnerung: call by reference

Damit der Wert von `r` in der Funktion `strtod` geändert werden kann, muss die Adresse von `r` übergeben werden!

```c
d = strtod(s, &r);   // richtig!!!!!!
```



Beispiel

```c
.....
int i;
long l;
char *r;
double d;

for (i = 1; i < argc; i += 2) {
    if (strcmp(argv[i], "-d") == 0) {
        d = strtod(argv[i + 1], &r);
        printf("%d. Parameter: %lf\n", i+1, d);
    }
    else if (strcmp(argv[i], "-l") == 0) {
        l = strtol(argv[i + 1], &r, 0);
        printf("%d. Parameter: %ld\n", i+1, l);
    }
    printf("Nicht umgewandelt: %s\n", r);
}
.....
```

Beispiel

Obiges Programm liefert bei dem Aufruf

```
strUmw -d 12.3ab -l 789.1cd -l 0X23AGH -l 022.2xy
```

die folgende Ausgabe:

```
2. Parameter: 12.300000
Nicht umgewandelt: ab
4. Parameter: 789
Nicht umgewandelt: .1cd
6. Parameter: 570
Nicht umgewandelt: GH
8. Parameter: 18
Nicht umgewandelt: .2xy
```

base = 0 → übliche Interpretation für int-Konstanten

Einfache Umwandlung von Zeichenketten

`double atof(const char *s)`

- wandelt s in double um
- analog zu `strtod(s, (char **)NULL)`

`int atoi(const char *s)`

- wandelt s in int um
- analog zu `(int)strtol(s, (char **)NULL, 10)`

`int atol(const char *s)`

- wandelt s in long um
- analog zu `strtol(s, (char **)NULL, 10)`

Beispiel

```
.....
int main(int argc, char *argv[]) {
    int i, *arr;

    argc -= 1;
    arr = (int *) malloc(argc * sizeof(int));
    for (i = 0; i < argc; i++)
        arr[i] = atoi(argv[i + 1]);

    sort(arr, argc);

    for (i = 0; i < argc; i++)
        printf("%d\n", arr[i]);

    return 0;
}
```


Übersicht

- Dateioperationen
- Hilfsfunktionen
- *Zeichenketten*
- Zufallszahlen

Funktionen für Zeichenketten: string.h

`char *strcpy(char *s, const char *ct)`
Zeichenkette ct in Vektor s kopieren, liefert s

`char *strncpy(char *s, const char *ct, int n)`
maximal n Zeichen aus ct in Vektor s kopieren, liefert s

`char *strcat(char *s, const char *ct)`
Zeichenkette ct an Zeichenkette s anhängen, liefert s

`char *strncat(char *s, const char *ct, int n)`
höchstens n Zeichen von ct an Zeichenkette s anhängen und mit \0 abschließen, liefert s

Funktionen für Zeichenketten

`int strcmp(const char *cs, const char *ct)`
Zeichenketten cs und ct vergleichen:

- liefert Wert < 0 wenn cs $<$ ct,
- liefert 0 wenn cs == ct und
- liefert Wert > 0 wenn cs $>$ ct.

`int strncmp(const char *cs, const char *ct, int n)`
höchstens n Zeichen von cs mit ct vergleichen

`char *strchr(const char *cs, int c)`
liefert Zeiger auf das erste c in cs, oder NULL

`char *strrchr(const char *cs, int c)`
liefert Zeiger auf das letzte c in cs, oder NULL

Funktionen für Zeichenketten

`char *strstr(const char *cs, const char *ct)`
liefert Zeiger auf erste Kopie von ct in cs, oder NULL.

`size_t strlen(const char *cs)`
liefert die Länge von cs (ohne '\0')

Funktionen für Zeichenketten

`char *strerror(int n)`
liefert Zeiger auf Zeichenkette, die in der Implementierung für Fehler n definiert ist. Anwendung: `strerror(errno)`

Beispiel:

```
for (i = 0; i < 5; i++)
    printf("%s\n", strerror(i));
```

liefert:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

Beispiel

`grep [-i] pattern file`
→ print lines matching [ignore-case] a pattern

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

char *uppercase(char *word) {
    int i, len;

    len = strlen(word);
    for (i = 0; i < len; i++)
        word[i] = toupper(word[i]);

    return word;
}
```


Beispiel

```
#define TRUE 1
#define FALSE 0
#define N 80

int main(int argc, char **argv) {
    FILE *file;
    int cnt, idx;
    char ignoreCase, pattern[N], line[N];

    /* Aufruf formal korrekt? */
    if ((argc != 3) && (argc != 4)) {
        printf("try: %s [-i] pattern file\n",
               argv[0]);
        return 1;
    }
```


Beispiel

```
    /* ignore-case? */
    if (strcmp(argv[1], "-i")) {
        ignoreCase = FALSE;
        idx = 2;
        strcpy(pattern, argv[1]);
    } else {
        ignoreCase = TRUE;
        idx = 3;
        strcpy(pattern, uppercase(argv[2]));
    }

    file = fopen(argv[idx], "r");
    if (file == NULL) {
        perror(argv[idx]);
        return 2;
    }
```


Beispiel

```
    /* Zeilenweise testen */
    cnt = 0;
    while (fgets(line, N, file)) {
        cnt += 1;
        if (ignoreCase) {
           if (strstr(uppercase(line), pattern))
              printf("%3d: %s", cnt, line);
        } else if (strstr(line, pattern))
           printf("%3d: %s", cnt, line);
    }

    fclose(file);
    return 0;
}
```


Funktionen für Zeichenketten

`char *strtok(char *s, const char *ct)`
durchsucht `s` nach Zeichenfolgen, die durch Zeichen aus `ct` begrenzt sind.

- Der erste Aufruf findet die erste Zeichenfolge in `s`, die nicht aus Zeichen in `ct` besteht.
- Die Folge wird abgeschlossen, indem das nächste Zeichen in `s` mit `\0` überschrieben wird.
- Resultat: Zeiger auf die Zeichenfolge.
- Bei jedem weiteren Aufruf wird `NULL` anstelle von `s` übergeben. Solch ein Aufruf liefert die nächste Zeichenfolge, wobei unmittelbar nach dem Ende der vorhergehenden Suche begonnen wird.

Hinweis: Die Zeichenkette `ct` kann bei jedem Aufruf verschieden sein.

Beispiel

Abspeichern der Studentendaten als CSV-Datei:

```
Huber,Hans Walter,23,5,1978,3,12345678
Meier,Ulrike Maria,12,8,1977,4,07654321
```

Dazu muss die Methode `readFromFile()` angepasst werden:

```
stud_t *readFromFile(char *fname, int *len) {
   stud_t *s;
   FILE *f;
   int i;
   char *str, line[200];

   f = fopen(fname, "r");
   if (f == NULL) {
       perror(fname);
       return NULL;
   }
```


Beispiel

```
   /* Anzahl Daten aus Header lesen */
   fgets(line, 200, f);
   *len = atoi(line);

   /* dynamisches Array anlegen */
   s = (stud_t *) malloc(*len * sizeof(stud_t));

   for (i = 0; i < *len; i++) {
       fgets(line, 200, f);

       /* Name und Vorname extrahieren */
       str = strtok(line, ",:;");
       strncpy(s[i].name, str, 30);
       str = strtok(NULL, ",:;");
       strncpy(s[i].vname, str, 30);
```


Beispiel

```
       /* Geburtsdatum extrahieren */
       str = strtok(NULL, ",:;");
       s[i].gebdat.tag = atoi(str);
       str = strtok(NULL, ",:;");
       s[i].gebdat.monat = atoi(str);
       str = strtok(NULL, ",:;");
       s[i].gebdat.jahr = atoi(str);

       /* Fachbereich und Matrikelnummer */
       str = strtok(NULL, ",:;");
       s[i].fb = atoi(str);
       str = strtok(NULL, ",:;");
       s[i].matrikelnr = atoi(str);
   }

   fclose(f);
   return s;
} /* End: readFromFile() */
```

Vollständiges Beispiel

Schneller Zugriff durch Index: erster Buchstabe des Namens
Voraussetzung: sortierte Datensätze, keine Umlaute

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LENGTH 100

int len[26] = {0};

int createIndex(char *filename);
void printStudents(char c, char *filename);
int getIndex(char c);
void toScreen(char *data);
```


Vollständiges Beispiel

```
void main(void) {
   char c;

   createIndex("_student.txt");
   printf("erster Buchstabe des Namens: ");
   scanf("%c", &c);
   printStudents(c, "_student.txt");
}

int getIndex(char c) {
   int e, i, idx = 0;

   e = tolower(c) - 'a';
   for (i = 0; i < e; i++)
       idx += len[i];
   return idx;
}
```


Vollständiges Beispiel

```
int createIndex(char *filename) {
   char line[LENGTH];
   FILE *f;

   f = fopen(filename, "r");
   if (f == NULL) {
       perror(filename);
       return -1;
   }

   while (fgets(line, LENGTH, f)) {
       char c = tolower(line[0]);
       len[c - 'a'] += strlen(line);
   }
   return fclose(f);
}
```


Vollständiges Beispiel

```
void printStudents(char c, char *filename) {
   FILE *f;
   char line[LENGTH];
   int idx = getIndex(c);

   f = fopen(filename, "r");
   if (f == NULL) {
       perror(filename);
       return -1;
   }

   fseek(f, idx, SEEK_SET); // Fehlerbehandlung?

   while (fgets(line, LENGTH, f)
          && toupper(line[0]) == toupper(c))
       toScreen(line);
   fclose(f);
}
```


Vollständiges Beispiel

```
void toScreen(char *data) {
   char name[30], vname[30];
   int tag, monat, jahr;
   short fb;
   long matrikelnr;

   strcpy(name, strtok(data, ";,:"));
   strcpy(vname, strtok(NULL, ";,:"));
   tag = atoi(strtok(NULL, ";,:"));
   monat = atoi(strtok(NULL, ";,:"));
   jahr = atoi(strtok(NULL, ";,:"));
   fb = atoi(strtok(NULL, ";,:"));
   matrikelnr = atol(strtok(NULL, ";,:"));

   printf("%s, %s, %02d.%02d.%4d, %hd, %ld\n",
          name, vname, tag, monat, jahr, fb,
          matrikelnr);
}
```


Übersicht

- Dateioperationen
- Hilfsfunktionen
- Zeichenketten
- *Zufallszahlen*

Pseudo-Zufallszahlen

Die Kennzeichen einer Zufallsfolge sind:

- Zahlen entstammen einem gegebenen Zahlenbereich.
- Zahlen sind unabhängig voneinander: aus der Kenntnis der ersten `n` Zahlen kann nicht auf die `(n+1)`-te Zahl geschlossen werden.
- Zahlen unterliegen gegebener Häufigkeitsverteilung.

Beispiel: Würfel

- Zahlen im Bereich 1 bis 6
- Gleichverteilung: Würfelt man genügend häufig, dann zeigt sich, dass die Zahlen 1 bis 6 ungefähr gleich oft erscheinen.

Pseudo-Zufallszahlen

Einfache Zufallszahlen zwischen 0 und m:

- $x_0 \in [0 \ldots m-1]$ wird vorgegeben $\rightarrow$ seed
- $x_{n+1} = (a \cdot x_n + 1) \bmod m$

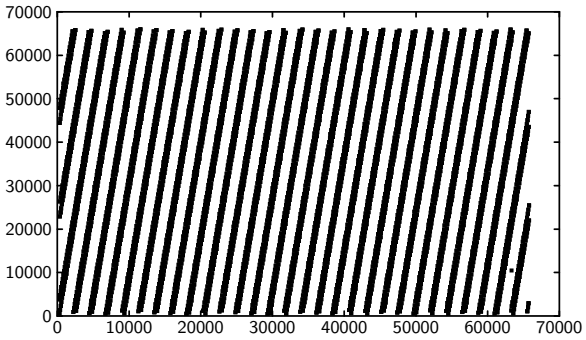
Beispiel: $x_{n+1} = (3423 \cdot x_n + 1) \bmod 2^{16}$

Die ersten 42 Werte für $x_0 = 0$:

0	1	3424	54945	53952	62785	20512
23521	34176	2689	29408	289	6208	16321
30112	50785	35584	38145	22624	43937	56768
2625	6944	45281	4224	40833	48608	54817
9024	21697	16544	7009	5632	10753	41824
32929	59584	8001	58912	1505	39808	13441

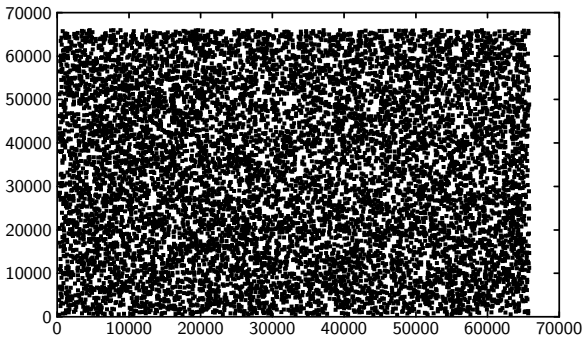
Pseudo-Zufallszahlen

Güte: Stelle je zwei Zufallszahlen als Koordinaten x, y dar. Bei *guter* Gleichverteilung ist das resultierende Bild gleich dicht mit Punkten gefüllt. → obige Folge ist schlecht!



Pseudo-Zufallszahlen

Gut: modifizierte Zufallsfolge $x_{n+1} = (3421 \cdot x_n + 1) \bmod 2^{16}$ mit $x_0 = 0$.



Pseudo-Zufallszahlen

Das obige graphische Verfahren soll das Problem der Güte von Zufallszahlen-Generatoren nur veranschaulichen.

Gute Generatoren sollten folgende Kriterien erfüllen:

- große Periode
- geringe Korrelation
- gleichmäßige Verteilung
- lange sich nicht überschneidende Teilfolgen
- Portierbarkeit
- Wiederholbarkeit

Aus der Mathematik* sind geeignete Tests für die Güte von Zufallszahlengeneratoren bekannt.

- Entropietest
- Chi-Quadrat-Test

* siehe auch D.E. Knuth: The art of computer programming, Volume 2.

Pseudo-Zufallszahlen in C

`int rand(void)`

- liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX` (mindestens $32767 = 2^{15} - 1$, ebenfalls definiert in `stdlib.h`)

`void srand(unsigned int seed)`

- benutzt `seed` als Ausgangswert für eine neue Folge von Pseudo-Zufallszahlen (entspricht x_0)

Die Güte des in C verwendeten Zufallszahlengenerators ist für *normale* Anwendungen ausreichend.

Beispiel

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10000

int main(void) {
   int i, arr[N];
   int seed = 1;

   srand(seed);
   for (i = 0; i < N; i += 2) {
       arr[i] = rand() % 200;
       arr[i + 1] = rand() % 200;
       printf("%3d, %3d\n", arr[i], arr[i + 1]);
   }
   printf("\n");

   return 0;
}
```