Die Programmiersprache C

Die Standardbibliothek

Die Standardbibliothek

Was ist das?

- Sammlung oft benötigter Funktionen/Prozeduren für
  * Ein- und Ausgabe,
  * Tests für Zeichenklassen, String-Bearbeitung,
  * mathematische Funktionen, ...
- Gutes Beispiel für prozedurale Programmierung sowie wiederverwendbarer Software.

Funktionen, Typen und Makros sind in Definitionsdateien deklariert:

`<ctype.h>` `<errno.h>` `<float.h>` `<limits.h>` `<math.h>`
`<stdarg.h>` `<stdio.h>` `<stdlib.h>` `<string.h>` `<time.h>`

Die Standardbibliothek (2)

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

Eine Definitionsdatei muss außerhalb von allen externen Vereinbarungen eingefügt werden, bevor irgendetwas benutzt wird, das in der Header-Datei vereinbart wird.

Hinweis: Für die Standardbibliothek sind externe Namen reserviert, die mit einem Unterstrich _ und

- einem Großbuchstaben oder
- einem weiteren Unterstrich beginnen.

Ein- und Ausgabe: `stdio.h`

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 (2)

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.

Ein- und Ausgabe: 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)
- `r+` zum Ändern öffnen
- `w+` zum Ändern erzeugen, alten Inhalt wegwerfen

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

Ein- und Ausgabe: Dateioperationen (2)

```
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

Ein- und Ausgabe: Dateioperationen (3)

```
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 temporäre Datei mit Modus `wb+` (automatisches Löschen bei `fclose` oder normalem Programmende)
- liefert einen Datenstrom oder 0 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 (negativ im Fehlerfall)

Umwandlungsangabe beginnt mit `%` und enthält optional:

- Steuerzeichen (`%+d`)
- Angabe zur Feldbreite (`%7f`)
- Angabe zur Genauigkeit (`%7.2f`)
- Längenangabe (`%ld`, `%hd`)

Formatierte Ausgabe (2)

Steuerzeichen:

- `-` Ausrichtung linksbündig
- `+` Ausgabe immer mit Vorzeichen
- `0` Auffüllen mit führenden Nullen
- *Leerzeichen* ist das erste Zeichen kein Vorzeichen, so wird ein Leerzeichen vorangestellt
- `#` alternative Form der Ausgabe:
  - `o` die erste Ziffer ist 0
  - `x`, `X` einem Wert wird `0x`/`0X` vorangestellt
  - `e`, `f`, `g` Ausgabe enthält immer einen Dezimalpunkt
  - `g`, `G` Nullen am Ende werden nicht unterdrückt
  - Beispiel `%#X`: hexadezimale Ausgabe anstelle von `%d`

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

Formatierte Eingabe (2)

Format-Zeichenkette:

- Leerzeichen und Tabulatorzeichen werden ignoriert: `fscanf(f,"%d %d", &x, &y)` $\widehat{=}$ `fscanf(f,"%d%d", &x, &y)`
- Zeichen, die den nächsten Zeichen nach einem umgewandelten Argument entsprechen müssen: `fscanf(f,"%d,%d", &x, &y)` liest `42,43`, aber nicht `42 43`.
- Umwandlungsangaben: `%` gefolgt von
  - `*`: verhindert Zuweisung an ein Argument (optional)
  - einer Zahl: legt maximale Feldbreite fest (optional)
  - `h`, `l`, `L`: beschreibt die Länge des Ziels; `short`, `long` oder `long double` (optional)
  - einem Umwandlungszeichen

Formatierte Eingabe (3)

Ein Eingabefeld

- ist als Folge von Zeichen definiert, die keine Zwischenraumzeichen sind (Leerzeichen, Tabulator `\t`, Zeilenumbruch `\n`, Seitenvorschub `\f`, ...)
- reicht bis zum nächsten Zwischenraumzeichen, oder bis eine explizit angegebene Feldbreite erreicht ist.

Hinweis: `fscanf` liest über Zeilengrenzen hinweg, um seine Eingabe zu finden.

Formatierte Eingabe: Beispiele

Format	Eingabe	Resultat
`%d%d`	`42 43`	`42 43`
	`42a 43`	`42 %`
	`42 43a`	`42 43`
`%d,%d`	`42,43`	`42 43`
	`42, 43`	`42 43`
	`42, a43`	`42 %`
	`42, 43a`	`42 43`
`%da,%d`	`42a,43`	`42 43`
	`42b,43`	`42 %`

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 Null, wenn für `file` ein Dateiende notiert ist.
- `int ferror(FILE *file)` liefert einen Wert ungleich Null, 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 (2)

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

Wenn die Datei `xyz.txt` nicht gefunden wird, erhalten wir bspw. die Ausgabe: `xyz.txt: No such file or directory`

Ein- und Ausgabe: Beispiel

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

Problem: Da die Anzahl der Datensätze in der Datei nicht bekannt ist, implementieren wir zunächst eine lineare Liste, der besseren Übersicht halber in einem eigenen Modul.

Datei `liste.h:`

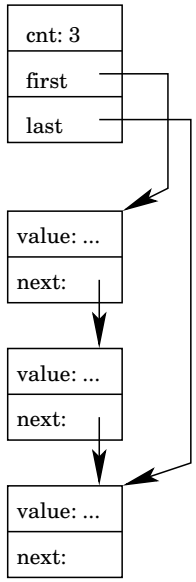
```
typedef struct {
    char *name, *vorname;
    short alter, fb;
    long matrikelnr;
} student_t;
```

```
typedef struct elem {
    student_t value;
    struct elem *next;
} listElem_t;
...
```


Ein- und Ausgabe: Beispiel (2)

```
typedef struct {
    int cnt;
    listElem_t *first;
    listElem_t *last;
    listElem_t *pos;
} liste_t;

liste_t *createList(void);
void append(liste_t *l, student_t s);
int getNext(liste_t *l, student_t *s);
void reset(liste_t *l);
```



Ein- und Ausgabe: Beispiel (3)

`liste.c:`

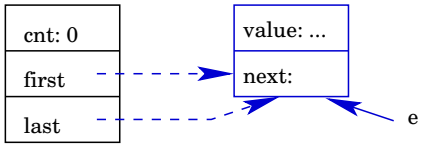
```
#include <stdlib.h>
#include "liste.h"

liste_t *createList(void) {
    liste_t *l = (liste_t *) malloc(sizeof(liste_t));
    l->cnt = 0;
    l->first = NULL;
    l->last = NULL;
    l->pos = NULL;
    return l;
}
...
```

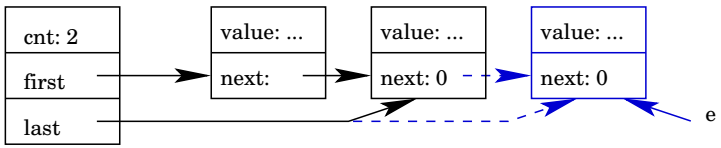

Ein- und Ausgabe: Beispiel (4)

Wenn ein neues Element an die Liste angehängt werden soll, sind zwei Fälle zu unterscheiden:

1. Fall: `cnt == 0`



2. Fall: `cnt > 0`



Ein- und Ausgabe: Beispiel (5)

```
void append(liste_t *l, student_t s) {
    listElem_t *e;
    e = (listElem_t *) malloc(sizeof(listElem_t));
    e->value = s;
    e->next = NULL;
    if (l->cnt == 0) {
        l->first = l->last = l->pos = e;
    } else {
        l->last->next = e;
        l->last = e;
    }
    l->cnt += 1;
}
```


Ein- und Ausgabe: Beispiel (6)

```
int getNext(liste_t *l, student_t *s) {
    if (l->pos == NULL)
        return 0;

    *s = l->pos->value;
    l->pos = l->pos->next;
    return 1;
}

void reset(liste_t *l) {
    l->pos = l->first;
}
```


Ein- und Ausgabe: Beispiel (7)

student.c:

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

void createStudent(student_t *s, char *name,
        char *vorname, short alter, short fb,
        long matrikelnr) {
    s->name = name;
    s->vorname = vorname;
    s->alter = alter;
    s->fb = fb;
    s->matrikelnr = matrikelnr;
}
...
```


Ein- und Ausgabe: Beispiel (8)

```
int saveToFile(student_t *s, int n, char *filename) {
    int i;
    FILE *f;

    f = fopen(filename, "w");
    if (f == NULL)
        return -1;

    for (i = 0; i < n; i++)
        fprintf(f, "%s,%s,%hd,%hd,%08ld\n",
            s[i].name, s[i].vorname, s[i].alter,
            s[i].fb, s[i].matrikelnr);

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

Ein- und Ausgabe: Beispiel (9)

```
liste_t *readFromFile(char *filename) {
    liste_t *l;
    student_t *s;
    FILE *f;
    char *name, *vorname;
    short alter, fb;
    long nr;

    f = fopen(filename, "r");
    if (f == NULL)
        return NULL;

    l = createList();
    ...
```


Ein- und Ausgabe: Beispiel (10)

```
    name = (char *) malloc(20 * sizeof(char));
    vorname = (char *) malloc(20 * sizeof(char));
    while (fscanf(f, "%5s,%4s,%hd,%hd,%ld",
                name, vorname, &alter, &fb, &nr) == 5) {
        s = (student_t *) malloc(sizeof(student_t));
        createStudent(s, name, vorname, alter, fb, nr);
        append(l, *s);
        name = (char *) malloc(20 * sizeof(char));
        vorname = (char *) malloc(20 * sizeof(char));
    }

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


Ein- und Ausgabe: Beispiel (11)

```
void main(void) {
   liste_t *l;
   student_t s[3], *p = s, t;

   createStudent(p++, "Huber", "Hans", 42, 3, 12345678);
   createStudent(p++, "Mayer", "Gabi", 37, 4, 7654321);
   createStudent(p++, "Meier", "Rosi", 34, 4, 98761234);

   saveToFile(s, 3, "_student.txt");
   l = readFromFile("_student.txt");

   while (getNext(l, &t))
      printf("%s, %s, %hd, %hd, %08ld\n", t.name,
             t.vorname, t.alter, t.fb, t.matrikelnr);
}
```


Ein- und Ausgabe: Fragen/Übungen

Es sollen Namen und Vornamen abgelegt werden, die beliebig lang sind (maximal 50 Zeichen). Es ist auch mehr als ein Vorname möglich. Die einzelnen Teile (Name, Vorname, Alter, FB, Matrikelnummer) sollen durch Kommata getrennt sein.

- Welches Problem tritt auf, wenn mit `fscanf` einzelne Strings eingelesen werden? Wie müsste die Funktion `readFromFile` umgeschrieben werden, um CSV-Dateien (Comma Separated Values) einzulesen?
- Schreiben Sie die `getNext`-Funktion so um, dass ein Zeiger auf `Student` zurückgegeben wird, oder Null bei Fehler. Wie ist die Funktion `main` umzuschreiben?

Antwort

Einlesen mittels

```
fscanf(file, "%s,%s,%hd,%hd,%ld", name, vname, ...)
```

funktioniert nicht:

- Beim Format `%s` wird das Komma mitgelesen und kann daher nicht als Trennsymbol verwendet werden.

  Einlesen von `Huber, Hans Walter, 42, 3, 08154711` wird als Namen `Huber,` liefern, anschließendes Komma wird nicht gefunden.
- Bei fester Feldbreite/Längenbeschränkung, bspw. `%50s` wird ggf. nur ein Vorname geliefert, da das Einlesen bei einem Leerzeichen abgebrochen wird.

Lösung: `readFromFile`

```
char teil[50];
...
while (fscanf(file, "%s", name) != EOF) {
    name[strlen(name) - 1] = '\0';  // Komma abschneiden
    vorname[0] = '\0';
    do {
        fscanf(file, "%s", teil);
        strcat(vorname, teil);    // concatenate strings
    } while (teil[strlen(teil) - 1] != ',');
    vorname[strlen(vorname) - 1] = '\0';
    fscanf(file, "%hd,%hd,%ld", &alter, &fb, &matr);
    ...
}
```


Lösung: `getNext`

```
student_t *getNext(liste_t *l) {
    student_t *s;

    if (l->pos == NULL)
        return NULL;

    s = &(l->pos->value);
    l->pos = l->pos->next;
    return s;
}
```


Lösung: `main`

```
void main(void) {
   liste_t *l;
   student_t s[3], *p = s;

   createStudent(p++, "Huber", "Hans", 42, 3, 12345678);
   createStudent(p++, "Mayer", "Gabi", 37, 4, 7654321);
   createStudent(p++, "Meier", "Rosi", 34, 4, 98761234);

   saveToFile(s, 3, "_student.txt");
   l = readFromFile("_student.txt");

   while ((p = getNext(l)) != NULL)
      printf("%s, %s, %hd, %hd, %08ld\n", p->name,
            p->vorname, p->alter, p->fb, p->matrikelnr);
}
```

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 0 bei Dateiende oder im Fehlerfall, ansonsten `s`

Ein- und Ausgabe von Zeichen (2)

`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`

Ein- und Ausgabe von Zeichen: 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';
}
```


Ein- und Ausgabe von Zeichen: Beispiel (2)

```
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;
    }
    ...
```

Ein- und Ausgabe von Zeichen: Beispiel (3)

```
    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;
}
```


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 (2)

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

Direkte Ein- und Ausgabe: Beispiel

```
int saveToFile(student_t *s, int n, char *filename) {
    int x;
    FILE *file;

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

    x = fwrite(s, sizeof(student_t), n, file);
    if (x != n)
        perror(filename);
    return fclose(file);
}
```

Direkte Ein- und Ausgabe: Beispiel (2)

```
int readFromFile(student_t *s, int n, char *filename) {
    int x;
    FILE *file;

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

    x = fread(s, sizeof(student_t), n, file);
    if (x != n)
        perror(filename);
    return fclose(file);
}
```


Direkte Ein- und Ausgabe: Beispiel (3)

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

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

    s = (student_t *) malloc(5 * sizeof(student_t));
    readFromFile(s, 5, filename);
    for (i = 0; i < 5; i++)
        printf("Student: %s %s, %hd, %hd, %ld\n",
                s[i].name, s[i].vorname,
                s[i].alter, s[i].fb, s[i].matrikelnr);

    return 0;
}
```


Direkte Ein- und Ausgabe (3)

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

```
typedef struct {
    char name[20];
    char vorname[20];
    short alter, fb;
    long matrikelnr;
} student_t;

void createStudent(student_t *s, char *name, char *vorn,
        short alter, short fb, long matrikelnr) {
    strncpy(s->name, name, 20);
    strncpy(s->vorname, vorn, 20);
    ...
}
```


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 Null verschiedenen Wert bei Fehler

`long ftell(FILE *file)`

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

Positionieren in Dateien (2)

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

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

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

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

Positionieren in Dateien: Beispiel

```
#include <stdio.h>
void main(void) {
    int z;
    char line[10];          /* feste Satzlänge: 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);
}
```


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 Null, 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 (2)

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

Hilfsfunktionen: `stdlib.h`

Umwandlung von Zeichenketten in Zahlen:

- `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`

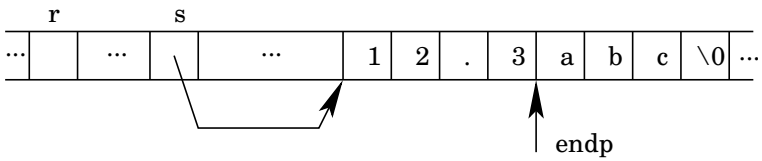
Die Funktionen speichern einen Zeiger auf den nicht umgewandelten Rest der Zeichenkette bei `*endp`.

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?

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

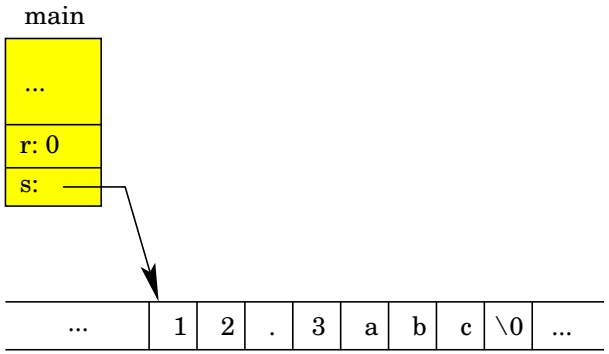


Damit der Wert von `r` in der Funktion `strtod` geändert werden kann, muss die Adresse von `r` übergeben werden.

```
d = strtod(s, &r);
```

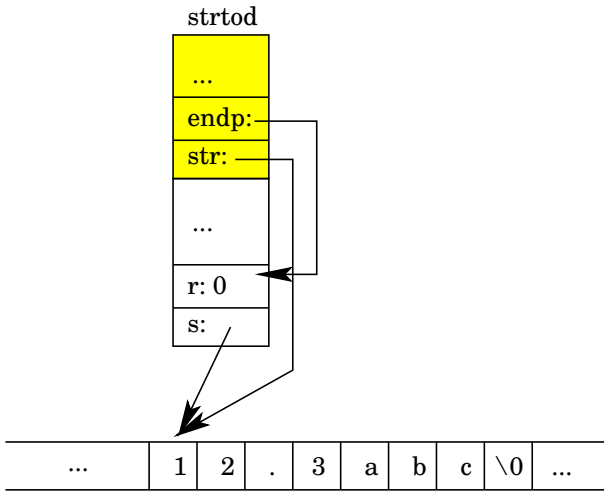

Erinnerung: call by reference (2)

Initial:



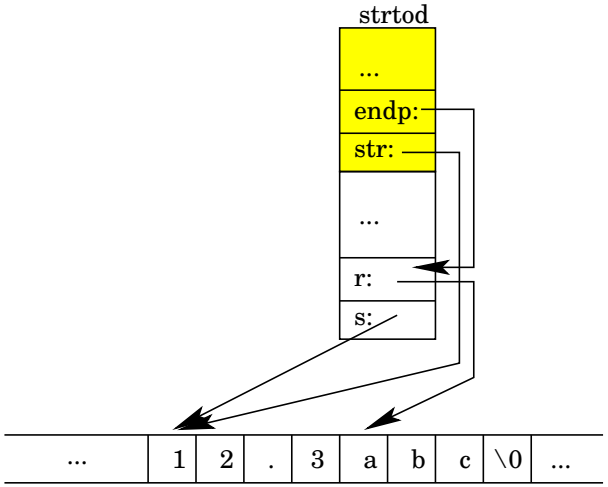
Erinnerung: call by reference (3)

Aufruf von `strtod`:



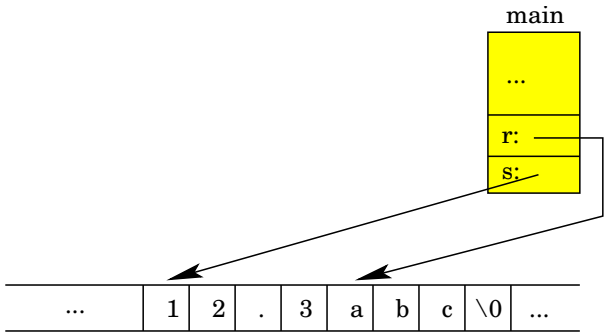
Erinnerung: call by reference (4)

vor dem Verlassen von `strtod`:



Erinnerung: call by reference (5)

Resultat:



Umwandlung von Zeichenketten: Beispiel

```
...
void main(int argc, char *argv[]) {
    int i;   char *r;   double d;   long l;

    for (i = 1; i < argc; i += 2) {
        if (strcmp(argv[i], "-d") == 0) {
            d = strtod(argv[i + 1], &r);
            printf("%d. Parameter: %f\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);
    }
}
```


Umwandlung von Zeichenketten: Beispiel (2)

Obiges Programm liefert bei dem Aufruf

```
strUmw -d 12.3abc -l 789.1abc -l 0X23AGH -l 022.2xy
```

die folgende Ausgabe:

```
2. Parameter: 12.300000
Nicht umgewandelt: abc
4. Parameter: 789
Nicht umgewandelt: .1abc
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)`

Umwandlung von Zeichenketten: Beispiel

```
...
void 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]);
}
```


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 (2)

Einfache Zufallszahlen zwischen 0 und m:

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

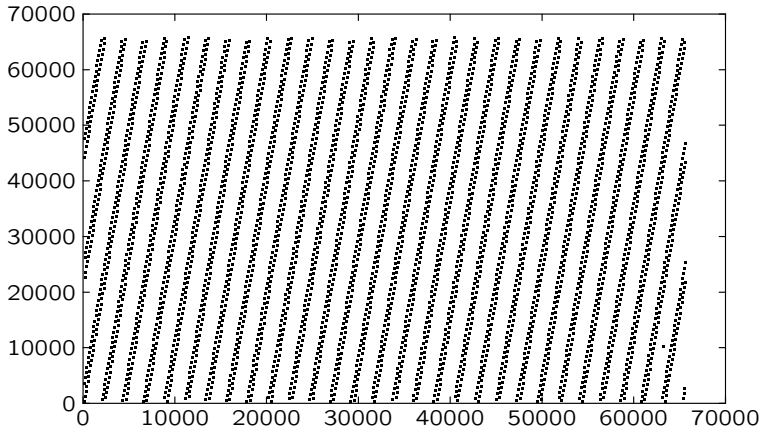
Beispiel: $x_{n+1} = (3423 \cdot x_n + 1) \bmod 2^{16}$

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

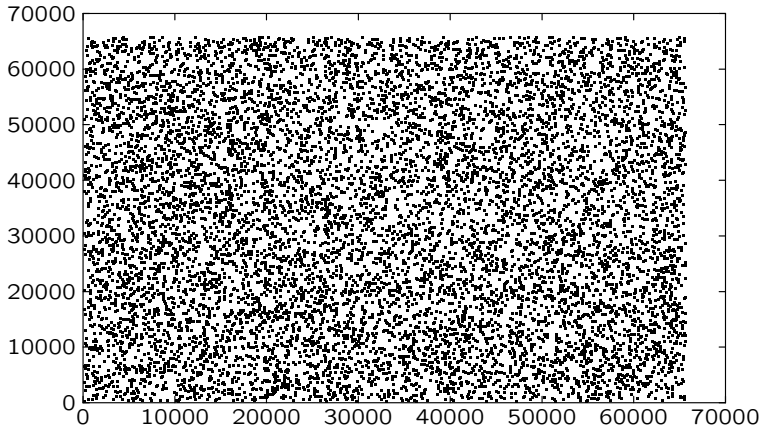
Pseudo-Zufallszahlen: Güte

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



Pseudo-Zufallszahlen: Güte (2)

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



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.

Pseudo-Zufallszahlen in C: Beispiel

```
...
#define N 10000

void 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");
}
```

Funktionen für Zeichenketten: `string`

`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 (2)

`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 (3)

`size_t strspn(const char *cs, const char *accept)`
liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich in `accept` vorkommen.

`size_t strcspn(const char *cs, const char *reject)`
liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich *nicht* in `reject` vorkommen.

`char *strpbrk(const char *cs, const char *ct)`
liefert Zeiger auf die Position in `cs`, an der irgendein Zeichen aus `ct` erstmals vorkommt, oder `NULL`.

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

Funktionen für Zeichenketten (4)

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

`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 < 30; i++)
    printf("%s\n",
        strerror(i));
```

liefert:

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

Funktionen für Zeichenketten: Beispiel

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

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

    len = strlen(*word);
    for (i = 0; i < len; i++)
        (*word)[i] = toupper((*word)[i]);
    return *word;
}
```


Funktionen für Zeichenketten: Beispiel (2)

```
#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;

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


Funktionen für Zeichenketten: Beispiel (3)

```
    /* 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)
        return 2;
```


Funktionen für Zeichenketten: Beispiel (4)

```
    /* Zeilenweise testen */
    cnt = 0;
    line = (char *) malloc(N * sizeof(char));
    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 (5)

`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.

Funktionen für Zeichenketten: Beispiel

Abspeichern der Studentendaten als CSV-Datei (Comma Separated Values).

```
Huber,Hans Walter,42,3,12345678
Meier,Ulrike Maria,37,4,07654321
```

Einlesen der Studentendaten aus CSV-Datei:

```
char line[100];
...
liste = createList();
while (fgets(line, 100, file)) {
    s = (Student *)malloc(sizeof(Student));
    extractStudent(s, line);
    insert(liste, *s);
}
```


Funktionen für Zeichenketten: Beispiel (2)

Extrahieren der Studentendaten (Name, Vorname, Alter, FB, Matrikelnummer) mittels `strtok`.

Umwandeln der Strings (Alter, FB, Matrikelnummer) in Zahlen mittels `atoi` bzw. `atol`.

```
void extractStudent(student_t *student, char *data) {
   char *s, *name, *vorn;
   short alter, fb;
   long matrikelnr;

   /* extrahieren der Daten aus String 'data' */
   s = strtok(data, ";,:");
   name = (char *)malloc((strlen(s)+1) * sizeof(char));
   strcpy(name, s);
```


Funktionen für Zeichenketten: Beispiel (3)

```
   s = strtok(NULL, ";,:");
   vorn = (char *)malloc((strlen(s)+1) * sizeof(char));
   strcpy(vorn, s);
   alter = atoi(strtok(NULL, ";,:"));
   fb = atoi(strtok(NULL, ";,:"));
   matrikelnr = atol(strtok(NULL, ";,:"));

   /* zuweisen der Werte an 'student' */
   student->name = name;
   student->vorname = vorn;
   student->alter = alter;
   student->fb = fb;
   student->matrikelnr = matrikelnr;
}
```

Vollständiges Beispiel

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

```
#define LENGTH 100
int len[26] = {0};
char line[LENGTH];

void main(int argc, char *argv[]) {
    char c, *filename = "_student.txt";

    createIndex(filename);
    printf("erster Buchstabe des Namens: ");
    scanf("%c", &c);
    printStudents(c, filename);
}
```


Vollständiges Beispiel (2)

```
int createIndex(char *filename) {
    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 (3)

```
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 (4)

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

    f = fopen(filename, "r");
    if (f == NULL) return;

    fseek(f, idx, SEEK_SET);   /* Fehlerbehandlung?? */
    while (fgets(line, LENGTH, f)
            && toupper(line[0]) == toupper(c))
        extractStudent(line);
    fclose(f);
}
```

Vollständiges Beispiel (5)

```
void extractStudent(char *data) {
    char name[20], vorname[20];
    short alter, fb;
    long matrikelnr;

    strcpy(name, strtok(data, ";,:"));
    strcpy(vorname, strtok(NULL, ";,:"));
    alter = atoi(strtok(NULL, ";,:"));
    fb = atoi(strtok(NULL, ";,:"));
    matrikelnr = atol(strtok(NULL, ";,:"));

    printf("%s, %s, %hd, %hd, %ld\n", name, vorname,
           alter, fb, matrikelnr);
}
```


Hilfsfunktionen: System

`void exit(int status)` beendet das Programm normal

- `atexit`-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung durchlaufen.
- Die Puffer offener Dateien werden geschrieben, offene Datenströme werden geschlossen.
- Die Kontrolle geht an die Umgebung des Programms zurück. 0 gilt als erfolgreiches Ende eines Programms.

`int atexit(void (*fcn)(void))` hinterlegt die Funktion `fcn`

- Liefert einen Wert ungleich 0, wenn die Funktion nicht hinterlegt werden konnte.

System: Beispiel

```
...
void exitFkt(void) {
    printf("Aufruf der atexit-Funktion!\n");
}

int main(int argc, char *argv[]) {
    int r;
    ...
    r = atexit(&exitFkt);
    if (r != 0) {
        printf("Exit-Funktion _NICHT_ hinterlegt!\n");
        exit(1);
    }
    ...
}
```


Hilfsfunktionen: System (2)

`int system(const char *s)`

- liefert die Zeichenkette `s` an die Umgebung
- die Umgebung führt die Anweisung `s` aus
- Resultat: Rückgabewert (Status) des auszuführenden Kommandos, im Fehlerfall -1. **Zur Erinnerung:** `main` kann einen Integer-Wert zurückliefern!

`char *getenv(const char *name)`

- liefert den Inhalt der Umgebungsvariablen `name`, im Fehlerfall `NULL` (falls keine Variable des Namens existiert)
- Details sind implementierungsabhängig

System: Beispiel

```
...
int main(int argc, char *argv[]) {
    int r;
    char *s;

    s = getenv("PATH");
    printf("%s\n", s);

    r = system("cp quelle ziel");
    if (r != 0) {
        printf("%d: cp _NICHT_ ausgeführt!\n", r);
        exit(1);
    }
    exit(0);
}
```


Mathematische Funktionen: `math.h`

In `math.h` vereinbarte Konstanten (Auszug):

`M_E`	2.7182818284590452354	e
`M_LOG2E`	1.4426950408889634074	$\log_2(e)$
`M_LOG10E`	0.43429448190325182765	$\log_{10}(e)$
`M_LN2`	0.69314718055994530942	$\log_e(2)$
`M_LN10`	2.30258509299404568402	$\log_e(10)$
`M_PI`	3.14159265358979323846	π
`M_PI_2`	1.57079632679489661923	$\pi/2$
`M_PI_4`	0.78539816339744830962	$\pi/4$
`M_1_PI`	0.31830988618379067154	$1/\pi$
`M_SQRT2`	1.41421356237309504880	$\sqrt{2}$
`M_SQRT1_2`	0.70710678118654752440	$1/\sqrt{2}$

Mathematische Funktionen (2)

Im weiteren gilt: `x,y: double` und `n: int`
Alle Funktionen liefern einen Wert vom Typ `double`.

`sin(x)`, `cos(x)`, `tan(x)`	Sinus, Cosinus, Tangens ...
`asin(x)`, `acos(x)`, `atan(x)`	... inverse Funktionen
`sinh(x)`, `cosh(x)`, `tanh(x)`	... hyperbolische Funktionen
`exp(x)`	e^x
`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$
`sqrt(x)`	$\sqrt{x}$ (Fehler: $x < 0$)
`pow(x, y)`	x^y (Fehler: $x < 0$ und $y \notin \mathbb{Z}$)
`fabs(x)`	Absolutwert (Betrag) von x
`ldexp(x, n)`	$x \cdot 2^n$

Mathematische Funktionen (3)

`ceil(x)` $\widehat{=} \lceil x \rceil$
kleinster ganzzahliger Wert, der nicht kleiner als x ist

`floor(x)` $\widehat{=} \lfloor x \rfloor$
größter ganzzahliger Wert, der nicht größer als x ist

`frexp(x, int *exp)`
zerlegt x in normalisierte Mantisse (Resultat) und in eine Potenz von 2 (wird in `*exp` abgelegt)

`modf(x, double *ip)`
zerlegt x in einen ganzzahligen Teil (wird bei `*ip` abgelegt) und in einen Rest (Resultat).

`fmod(x, y)` $\widehat{=} x - \lfloor \frac{x}{y} \rfloor \cdot y$

Mathematische Funktionen (4)

In `errno.h` befinden sich die Konstanten `EDOM` und `ERANGE`, die Fehler im Argument- und Resultatbereich der Funktionen anzeigen. **Wichtig: `errno` vor Funktionsaufruf löschen!**

```
errno = 0;
erg = sqrt(-2.0);
if (errno == EDOM)    /** Argumentfehler **/
    printf("-2.0: wrong argument for sqrt!\n");
else printf("sqrt(-2.0) = %f\n", erg);

errno = 0;
erg = exp(800);
if (errno == ERANGE)  /** Resultatfehler **/
    printf("exp(800): out of range!\n");
else printf("exp(800) = %f\n", erg);
```


Datum und Uhrzeit: `time.h`

`clock_t`, `time_t`: arithmetische Typen, repräsentieren Zeiten

`struct tm` enthält Komponenten einer Kalenderzeit:

`tm_sec`	Sekunden nach der vollen Minute
`tm_min`	Minuten nach der vollen Stunde
`tm_hour`	Stunden seit Mitternacht
`tm_mday`	Tage im Monat (1..31)
`tm_mon`	Monate **seit** Januar (0..11)
`tm_year`	Jahre **seit** 1900
`tm_wday`	Tage seit Sonntag (0..6)
`tm_yday`	Tage seit dem 1. Januar (0..365)
`tm_isdst`	Kennzeichen für Sommerzeit

Datum und Uhrzeit (2)

`time_t time(time_t *t)`
liefert die aktuelle Kalenderzeit, Resultat auch bei `*t`

`double difftime(time_t time2, time_t time1)`
liefert `time2 - time1` ausgedrückt in Sekunden

`struct tm *gmtime(const time_t *t)`
wandelt die Kalenderzeit `*t` in *Coordinated Universal Time* UTC (historisch: Greenwich Mean Time)

`struct tm *localtime(const time_t *t)`
wandelt die Kalenderzeit `*t` in Ortszeit

Datum und Uhrzeit (3)

```
#include <time.h>

void main() {
    time_t t;
    struct tm *d;

    t = time(0);          /* #Sekunden seit 1.1.1970 */
    d = localtime(&t);    /* Sekunden -> lokale Zeit */
    printf("Datum: %02d.%02d.%04d\n",
        d->tm_mday, d->tm_mon + 1, d->tm_year + 1900);
    printf("Zeit: %02d:%02d\n", d->tm_hour, d->tm_min);
    d = gmtime(&t);
    printf(" GMT: %02d:%02d\n", d->tm_hour, d->tm_min);
}
```

Datum und Uhrzeit (4)

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *t)
```

formatiert Datum und Zeit aus `*t` unter Kontrolle von `fmt` nach `s`, analog zu `sprintf`

`%a` `%A`	abgekürzter/voller Name des Wochentags
`%b` `%B`	abgekürzter/voller Name des Monats
`%c`	Datum und Uhrzeit
`%d`	Tag im Monat (1..31)
`%H`	Stunde (0..23)
`%j`	Tag im Jahr (1..366)
`%U`	Wochen im Jahr

Datum und Uhrzeit (5)

Zeitmessung im Programm:

```
...
int main(int argc, char **argv) {
    time_t time1, time2;
    double diff;
    ...
    time1 = time(0);
    /* zu messende Programmstelle */
    ...
    time2 = time(0);
    diff = difftime(time2, time1);
    ...
}
```


Datum und Uhrzeit (6)

Formatierung von Datum und Zeit:

```
void main() {
    char datum[80];
    time_t t;
    struct tm *d;

    t = time(0);
    d = localtime(&t);

    strftime(datum, 80, "%a, %d. %B %Y", d);
    printf("%s\n", datum);
}
```

liefert zum Beispiel: `Wed, 12. January 2005`

Datum und Uhrzeit (7)

Zeitmessung unter Unix/Linux: `gettimeofday`

```
#include <sys/time.h>
void main(int argc, char *argv[]) {
    long t;
    struct timeval t1, t2;
    ...
    gettimeofday(&t1, NULL);
    doSomething(args);
    gettimeofday(&t2, NULL);

    t  = t2.tv_sec * 1000000 + t2.tv_usec;
    t -= (t1.tv_sec * 1000000 + t1.tv_usec);
    printf("time: %f ms\n", ((float)t) / 1000);
}
```

Datum und Uhrzeit (8)

Zeitmessung unter Windows:

- `getSystemTime`
- `getSystemTimeAsFileTime`

Verzeichnisse: `dirent.h`

`DIR *opendir(const char *name)`
öffnet das angegebene Verzeichnis und liefert einen Datenstrom analog zu `fopen`.

`int closedir(DIR *dir)`
schließt den angegebenen Datenstrom analog zu `fclose`.

`struct dirent *readdir(DIR *dir)`
liefert einen Zeiger auf eine Struktur, die den nächsten Eintrag im Verzeichnis-Datenstrom repräsentiert.

`struct dirent`

`d_type`	Typ der Datei (Verzeichnis, Link, Datei, ...)
`d_name`	Name der Datei

Verzeichnisse: Beispiel (1)

Verzeichnis-Inhalt anzeigen: ohne Fehlerbehandlung!

```
void main(int argc, char *argv[]) {
   DIR *dir;
   struct dirent *entry;

   dir = opendir(argv[1]);
   while ((entry = readdir(dir)) != 0) {
      if (entry->d_type == 4)   /**  directory   **/
         printf("Verzeichnis: %s\n", entry->d_name);
      if (entry->d_type == 8)   /** regular file **/
         printf("Datei: %s\n", entry->d_name);
   }
   closedir(dir);
}
```


Verzeichnisse: Beispiel (2)

Verzeichnis- und Unterverzeichnisinhalt anzeigen:

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

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("usage: %s path\n", argv[0]);
        return 1;
    }

    getDirEntries(argv[1]);
    return 0;
}
```

Verzeichnisse: Beispiel (3)

```
void getDirEntries(char *dirname) {
    DIR *dir;
    char subdir[100];
    struct dirent *entry;

    dir = opendir(dirname);
    if (dir == NULL) {
        printf("could not open %s\n", dirname);
        return;
    }
    ...
```


Verzeichnisse: Beispiel (4)

```
    while ((entry = readdir(dir)) != NULL) {
        if ((strcmp(entry->d_name, ".") == 0)
                || (strcmp(entry->d_name, "..") == 0))
            continue;
        if (entry->d_type == 4) {  /** directory **/
            strcpy(subdir, dirname);
            strcat(subdir, "/");
            strcat(subdir, entry->d_name);
            getDirEntries(subdir);
        }
        if (entry->d_type == 8)   /** regular file **/
            printf("%s\n", entry->d_name);
    }
    closedir(dir);
}
```


Erweiterungen im C99 Standard

inline functions: Compiler-Hinweis, jeder Aufruf der Funktion ist durch Einfügen des Codes der Anweisungen des Funktionsrumpfs zu ersetzen.

Beispiel:

```
inline int max(int i, int j) {
    return (i > j) ? i : j;
}
```

⇒ spart das Erzeugen von Variablen sowie das Kopieren von Werten für Argumente und Funktionswert!

⇒ nur sinnvoll für Funktionen, deren Rumpf nur wenige Anweisungen enthalten

Erweiterungen im C99 Standard (2)

Variablen-Deklaration sind nun an vielen Stellen im Code erlaubt, nicht nur zu Beginn eines Blocks

Beispiel:

```
#include <stdio.h>
main() {
    for (int i = 0; i < 10; i++)
        printf("%2d: Hello, world!\n", i);

    int x = 1;
    while (x < 10)
        printf("x = %3d\n", x++);
}
```

⇒ kein sinnvolles Feature, Code wird schlechter lesbar

Erweiterungen im C99 Standard (3)

neue Datentypen:

- `long long int` schließt Lücke zwischen 32- und 64-Bit
- `complex` Darstellung komplexer Zahlen + Arithmetik

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

main() {
   complex c = 1.0f + 1.0f * _Complex_I;

   printf("c+c = %f+i%f\n", creal(c+c), cimag(c+c));
   printf("c-c = %f+i%f\n", creal(c-c), cimag(c-c));
   printf("c*c = %f+i%f\n", creal(c*c), cimag(c*c));
   printf("c/c = %f+i%f\n", creal(c/c), cimag(c/c));
}
```


Erweiterungen im C99 Standard (4)

erweiterte/neue Bibliotheken

- `stdint.h` Definition von `long long int`
- `complex.h` Darstellung komplexer Zahlen + Arithmetik
- `snprintf(char *str, size_t size, char *format, ...)` schreibt maximal `size` Zeichen nach `str`, Formatierung erfolgt anhand `format`
- `va_copy(va_list dest, va_list src)`

Erweiterungen im C99 Standard (5)

heute üblich: **Unicode**-Zeichensatz

- C90 definert **breite Zeichen**: Datentyp `wchar_t`
- C99 definiert alle Bibliotheksfunktionen über Zeichenketten auch in einer Version für breite Zeichen

```
#include <wchar.h>
int main(void) {
    wchar_t ustr[20] = L"abcde";
    int len;

    swprintf(ustr, L"%4i", 1234);
    len = wcslen(ustr);
    return 0;
}
```


Erweiterungen im C99 Standard (6)

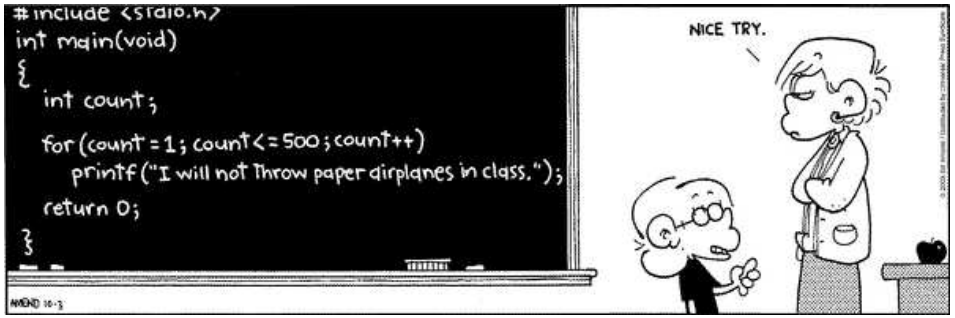
- **einzeilige Kommentare wie in C++:**

```
// Hauptprogramm
int main(int argc, char *argv[]) {
    int upper;  // obere Grenze
    int lower;  // untere Grenze
    ...
```

- **Arrays variabler Länge:**

```
int len = atoi(argv[1]);
int arr[len];    // nach ISO-C90 verboten
```

Was noch bleibt: Anwenden



Was noch bleibt: Programmentwicklung

Kenntnis der Programmiersprache: notwendige, aber nicht ausreichende Bedingung, um gute Software zu schreiben.

Software-Engineering:

- Ende der 60er Jahre: Software-Krise, Komplexität der Programme wuchs den Entwicklern über den Kopf, Fehlerbeseitigung führt zu neuen Fehlern.
- Erkenntnis: Software ist Ergebnis von Ingenieurtätigkeit
- Software ist industrielles Produkt mit definierten Qualitätsmerkmalen, das methodischer Planung, Entwicklung, Herstellung und Wartung bedarf.
- Es werden Methoden und Werkzeuge benötigt. (CASE-Tool: Computer Aided Software Engineering)

Programmentwicklung: Phasenmodell

Hier nur eine kurze Übersicht, näheres in der Vorlesung *Software-Engineering* bei Prof. Dr. Beims.

Lebenszyklus von Software: (software life cycle)

Problemanalyse: Zusammen mit dem Auftraggeber wird das Problem definiert und analysiert. Dient der genauen Spezifikation des Produkts.

Festlegen der Funktionalität, untersuchen der Durchführbarkeit, prüfen der ökonomischen Sinnhaftigkeit.

Resultat: Pflichtenheft, Benutzerhandbuch, Projekt- und Testplan.

Programmentwicklung: Phasenmodell (2)

Entwurfsphase: das Software-System wird als Ganzes entworfen und in Teile zerlegt (→ **Grobentwurf**), mit den Teilen ebenso verfahren (→ **Feinentwurf**).

Zerlegen in Teilprobleme, bis Aufgabe überschaubar:

- Das Programm ist die Gesamtheit der Module.
- Die Schnittstellen zwischen den Modulen müssen exakt definiert sein.

Auch unter Zeitdruck sind diese Tätigkeiten durchzuführen: **Je früher man mit dem Codieren beginnt, desto später ist man damit fertig.**

Denn: der Überblick geht verloren, die Aufgabe erscheint schwierig, Lösungen sind unnötig kompliziert.

Programmentwicklung: Phasenmodell (3)

Zerlegen in überschaubare Einheiten führt oft auf bereits gelöste Probleme (Top-Down-Methode). Die Analyse steht im Vordergrund. Geeignet für Entwicklung neuer Produkte.

Bottom-Up-Methode: Zusammenstellen von fertigen Bausteinen zu einem Produkt. Ungeeignet für Entwicklungsphase: verlieren in Details, Ziel wird aus dem Auge verloren. Geeignet für die Implementierungsphase.

Modularisierung:

- Eine Funktion führt eine abgeschlossene Aufgabe aus.
- Quellcode einer Funktion: nicht länger als zwei Seiten.
- Funktionen innerhalb einer Datei: wenn sie gemeinsame Daten verwenden oder logisch zusammengehören.

Programmentwicklung: Phasenmodell (4)

Implementieren: Editieren, Übersetzen, Binden

Testen: Es ist in der Regel nicht möglich, zu beweisen, dass ein Programm für alle möglichen Eingaben korrekt ist. **Testen des Programms zeigt nur die Anwesenheit von Fehlern, nicht deren Abwesenheit.**

Um sicherzustellen, dass die Funktionalität des Programms auch nach Änderungen/Erweiterungen gegeben ist, müssen Testdatensätze gesammelt und Tests geschrieben werden. **Tests müssen vollständig automatisiert sein und ihre Ergebnisse selbst überprüfen**, ansonsten testet niemand.

Programmentwicklung: Phasenmodell (5)

Wartungs- und Pflegephase: Programmierer sind 80% der Zeit mit Wartung und Pflege von Software beschäftigt.
⇒ keine Tricks anwenden, die andere nicht verstehen oder nachvollziehen können

Konstruktive Voraussicht: Mangelhafter Entwurf führt oft in Sackgassen, da sich neue Aspekte nicht realisieren lassen → weitsichtiger Entwurf und Implementierung

- trotz aller Voraussicht sind nicht alle Erweiterungen vorhersehbar
- oft werden erwartete Erweiterungen nicht realisiert, so dass der Entwurf unnötig komplex ist

⇒ **Refactoring**

Programmentwicklung: Was noch fehlt

Qualitätssicherung: Maßnahmen, deren Ziel die Verbesserung der Qualität des Software-Produktes ist.

Problem: Wie definiert man Qualität? Länge von Prozeduren, Schachtelungstiefe von Bedingungen und Schleifen, Struktur arithmetischer Ausdrücke, ... ???

Alle Herstellungsphasen des Produkts sind betroffen.

Programmentwicklung: Was noch fehlt (2)

Dokumentation: Jede Phase des Software-Projekts muss dokumentiert sein.

Problem: Dokumentation muss aktuell sein, sonst ist sie wertlos.

- Spezielle Kommentarformate im Source-Code können zur PDF- oder HTML-Dokumentation verarbeitet werden → javadoc, doxygen
- Programmstruktur ist mittels UML-, Modul- und Struktogrammen darstellbar
- die Datenbankstruktur kann als Entity-Relationship Diagramm beschrieben werden

Programmentwicklung: Erfolge

Methoden, die erfolgreich in der Programmentwicklung eingesetzt werden:

- Strukturierte Programmierung
- Top-Down-Entwurf
- Modulare Programmierung
- Objektorientierte Programmierung