Praktikum C-Programmierung 2026

IO

Jannek Squar

2026-06-03

Scientific Computing
Universität Hamburg

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Outline

Table of Contents

Motivation

- IEEE Standard: Portable Operating System Interface (POSIX)
- Ein-/Ausgabe
  - Konfigurationsdaten und Inputdateien
  - Berechnungsergebnisse
  - Bisher: `printf`
- Unterschiedliche Abstraktionsebenen
  - POSIX Ein-/Ausgabe mit Datei-Deskriptoren
  - Gepufferte Ein-/Ausgabe
  - Implizite Ein-/Ausgabe mittels Memory Mapping

open

```
int open(const char *pathname, int flags, mode_t mode);
```

```c
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  int fd;
  fd = open("myfile", O_RDWR | O_CREAT, 0644);
  assert(fd != -1);
  close(fd);
  return 0;
}
```

```
$ ./open.x; echo $?
0
$ ulimit -n
1024
```

- `creat`: `open` mit `O_CREAT|O_WRONLY|O_TRUNC` Flags
- Rückgabewert: Datei-Deskriptor
- Die Flags geben an, wie die Datei geöffnet werden soll (siehe `man open`)
- `close`: File-Deskriptor schließen (endliche Ressource, vgl. `ulimit -n`)

open ohne close

```c
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
  int fd;
  for(int i = 0; i < 2000; i++) {
    printf("i: %d\n", i);
    fd = open("myfile", O_RDWR | O_CREAT, 0644);
    assert(fd != -1);
  }
  return 0;
}
```

```
$ ./open_too-many.x
[...]
i: 1017
i: 1018
open_too-many.x: open_too-many.c:9: main: Assertion `fd != -1' failed.
[1]    93844 IOT instruction (core dumped)  ./open_too-many.x
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

```c
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
  int fd, value = 42;
  ssize_t nb;
  fd = open("myfile", O_WRONLY | O_CREAT, 0644);
  printf("Dateideskriptor: %d\n", fd);
  nb = write(fd, &value, sizeof(value));
  assert(nb == sizeof(value));
  close(fd);
  return 0;
}
```

```
$ ./write0.x
Dateideskriptor: 3
$ bat myfile
File: myfile   <BINARY>
```

- `write`: Schreibt Daten in die Datei
  - Binär, für Text vorherige Umwandlung in String notwendig
- Nicht notwendigerweise alle Daten auf einmal geschrieben
  - Rückgabewert muss überprüft und `write` u.U. erneut aufgerufen werden!

read v0

```
ssize_t read(int fd, void *buf, size_t count);
```

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int fd, value = 0;
  ssize_t nb;
  fd = open("myfile", O_RDONLY , 0644);
  nb = read(fd, &value, sizeof(value));
  assert(nb == sizeof(value));
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  close(fd);
  return 0;
}
```

```
$ ./read0.x
Bytes: 4, Inhalt: 42
```

- Mit `read` können Daten aus einer Datei gelesen werden
  - Funktioniert analog zu `write`, es wird ein Bytestrom gelesen
- Es werden wieder nicht notwendigerweise alle Daten gelesen

read v1

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int fd, value = 0;
  ssize_t nb;
  fd = open("myfile", O_RDONLY, 0644);
  nb = read(fd, &value, sizeof(value));
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  value = 0;
  nb = read(fd, &value, sizeof(value));
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  close(fd);
  return 0;
}
```

```
$ ./read1.x
Bytes: 4, Inhalt: 42
Bytes: 0, Inhalt: 0
```

- Jede Lese- oder Schreiboperation verändert den Dateizeiger
  - Dateizeiger ist dem Dateideskriptor zugeordnet
- Ein Rückgabewert von `0` zeigt das Dateiende an

```
off_t lseek(int fd, off_t offset, int whence);
```

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int fd, value= 0;
  ssize_t nb;
  fd = open("myfile", O_RDONLY, 0644);
  nb = read(fd, &value, sizeof(value));
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  lseek(fd, 0, SEEK_SET);
  nb = read(fd, &value, sizeof(value));
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  close(fd);
  return 0;
}
```

```
$ ./lseek.x
Bytes: 4, Inhalt: 42
Bytes: 4, Inhalt: 42
```

- Absolut: (SEEK_SET)
- relativ: (SEEK_CUR)
- relativ zum Ende: (SEEK_END)

pread

```
ssize_t pread(int fd, void *buf, size_t count, off_t
offset);
```

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int fd, value = 0;
  ssize_t nb;
  fd = open("myfile", O_RDWR | O_CREAT, 0644);
  nb = pread(fd, &value, sizeof(value), 0);
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  nb = pread(fd, &value, sizeof(value), 0);
  printf("Bytes: %zd, Inhalt: %d\n", nb, value);
  close(fd);
  return 0;
}
```

```
$ ./pread.x
Bytes: 4, Inhalt: 42
Bytes: 4, Inhalt: 42
```

- `pread` und `pwrite` erlauben Angabe von Offsets
  - Dateizeiger wird nicht verändert
  - Datei-Deskriptor von mehreren Threads gleichzeitig nutzbar

write v1

```c
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  write(STDOUT_FILENO, "Hallo Welt!\n", 13);
  return 0;
}
```

```
$ ./write1.x
Hallo Welt!
```

- Ein-/Ausgabe im Terminal findet auch über Dateideskriptoren statt
- Standardeingabe, -ausgabe und -fehlerausgabe sind standardmäßig geöffnet
  - Belegen üblicherweise die Dateideskriptoren `0` bis `2`
  - Zugriff über `STDIN_FILENO`, `STDOUT_FILENO` und `STDERR_FILENO`

Robustes Schreiben

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
    int fd;
    char text[] = "Hallo Welt\n";
    char *ptr = text;
    size_t remaining = strlen(text);

    fd = open("myfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    assert(fd != -1);

    while (remaining > 0) {
        ssize_t written = write(fd, ptr, remaining);
        assert(written >= 0);
        ptr += written;
        remaining -= written;
    }
    close(fd);
    return 0;
}
```

```
int stat(const char *restrict pathname, struct stat *restrict statbuf);
```

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <time.h>

int main(void) {
  int ret;
  struct stat st;
  ret = stat("myfile", &st);
  assert(ret == 0);
  printf("Mode: %o\n", st.st_mode);
  printf("File size: %ld bytes\n", st.st_size);
  printf("Last mod.: %s", ctime(&st.st_mtime));
  return 0;
}
```

```
$ ./stat.x
Mode: 100644
File size: 4 bytes
Last mod.: Mon Jun  1 15:47:02 2026
```

- Mit `stat` können Informationen über Dateien abgerufen werden
  - U. a. die Berechtigungen, der Besitzer, die Größe und diverse Zeitstempel

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main(void) {
  int fd;
  struct stat st;
  fd = open("myfile", O_RDWR | O_CREAT, 0644);
  lseek(fd, 1000000, SEEK_SET);
  assert(write(fd, &fd, sizeof(fd)) == sizeof(fd));
  assert(fstat(fd, &st) == 0);
  printf("Groesse: %lu\n", st.st_size);
  printf("Echte Groesse: %lu\n", st.st_blocks * 512);
  close(fd);
  return 0;
}
```

```
$ ./sparse.x
Groesse: 1000004
Echte Groesse: 8192
$ ls -l myfile
-rw-r--r-- 1 squar squar 1000004 Jun  1 16:17 myfile
$ du -h myfile
8.0K  myfile
```

- `stat` nimmt Dateipfad, `fstat` Dateideskriptor als Argument
- Sparse Dateien belegen nicht den gesamten Platz
  - Leere Bereiche liefern `0`en zurück

```c
#include <assert.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
  char dummy[999996];
  int fd;
  ssize_t nb;
  fd = open("myfile", O_RDWR | O_CREAT, 0644);
  nb = pwrite(fd, dummy, sizeof(dummy), sizeof(fd));
  assert(nb == sizeof(dummy));
  fsync(fd);
  close(fd);
  return 0;
}
```

- Nach `write` befinden sich Daten noch nicht auf dem Speichergerät
  - Üblicherweise in einem Cache des Betriebssystems
  - Caching kann mit `O_DIRECT` vermindert werden
    - Keine garantierte Unterstützung
    - Caching meist gut für Performance

Table of Contents

- Ausgewählte Funktionen:
  - `FILE *fopen(const char *restrict pathname, const char *restrict mode)`
  - `size_t fwrite(const void *ptr, size_t size, size_t n, FILE *restrict stream)`
  - `size_t fread(void *ptr, size_t size, size_t n, FILE *restrict stream)`
  - `int fclose(FILE *stream)`
- Durch Pufferung u. U. explizites `fflush` notwendig
- Binär, für Text kann `fprintf` benutzt werden

Gepufferte Ein-/Ausgabe II

- Benutzung etwas komfortabler, da „keine“ Flags notwendig sind
  - Modus wird als String angegeben
  - `"r"` ≡ `O_RDONLY`
  - `"w"` ≡ `O_WRONLY | O_CREAT | O_TRUNC`
  - `"a"` ≡ `O_WRONLY | O_CREAT | O_APPEND`
  - `"r+"` ≡ `O_RDWR`
  - `"w+"` ≡ `O_RDWR | O_CREAT | O_TRUNC`
  - `"a+"` ≡ `O_RDWR | O_CREAT | O_APPEND`

Beispiel

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

int main(void) {
    FILE *fp;
    size_t nb;
    int value = 42;

    fp = fopen("myfile_buffered", "wb");
    assert(fp != NULL);
    printf("Dateizeiger: %p\n", fp);
    nb = fwrite(&value, sizeof(value), 1, fp);
    assert(nb == 1);
    fclose(fp);

    value = 0;

    fp = fopen("myfile_buffered", "rb");
    assert(fp != NULL);
    nb = fread(&value, sizeof(value), 1, fp);
    printf("Elemente: %zu, Inhalt: %d\n", nb, value);
    fclose(fp);
    return 0;
}
```

```
$ ./buffered_io.x
Dateizeiger: 0x3d95e310
Elemente: 1, Inhalt: 42
```

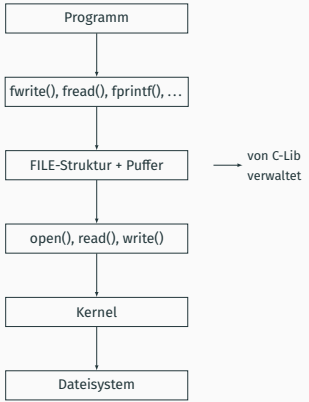

Table of Contents

Motivation

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- Abbildung einer Datei in den Arbeitsspeicher
  - Zugriff wie auf ein Array
  - Änderungen wie bei normalen Speicherbereichen, z. B. mit `memcpy`
  - Lesen und Schreiben wird vom Betriebssystem übernommen
  - Effiziente Verarbeitung großer Dateien

Beispiel

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

int main(void) {
  int fd = open("myfile", O_RDWR);
  int *data = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  printf("Alter Wert: %d\n", *data);
  *data = 9001;

  munmap(data, sizeof(int));
  close(fd);
  return 0;
}
```

```
$ ./write0.x
Dateideskriptor: 3
$ ./mmap2.x
Alter Wert: 42
$ ./read0.x
Bytes: 4, Inhalt: 9001
```