C: Input/Output

Gliederung

Wiederholung

Geschichte von Terminals

termios/ioctl & terminfo/termcap

Dateihandling

Pointer

Variablen im Speicher

- Jeder Variable kann eine Adresse zugeordnet werden, an der diese im Speicher liegt (mit Ausnahmen)
- Man sagt diese Adresse „zeigt“ auf diese Variable
- In C gibt es Pointer, welche wiederum Variablen sind, die eine solche Adresse als Wert halten
  - Insbesondere können diesen Variablen *ebenfalls* Adressen zugeordnet werden! Also Adressen die auf Adressen zeigen.
- Diese Pointer spezifizieren zusätzlich den Typ auf den sie zeigen

Pointerarithmetik

Der Typ auf den diese Variablen zeigen ist wichtig!

- Pointerarithmetik ist so definiert, dass ein inkrement um *n* in einen Pointer resultiert, der *n Elemente* weiter zeigt
- D. i. nicht per se die Adresse um *n* inkrementiert! Bspw.:

```
1 int *p = 0xf00;
2 // ^\ p zeigt auf den int an Adresse 0xf00
3 int *q = p+4;
4 // ^\ p zeigt auf den int an Adresse 0xf00 + 4*sizeof (int)
```

Matrixaufgabe

Automatische Variable:

```
1 // allokiert Speicher vom Stack
2 // ! kann überlaufen, keine Fehlerbehandlung
3 int M[3][6];
4 /**
5  * Bild im Speicher:
6  * M:   [iiiiii|iiiiii|iiiiii]
7  *       \____/ \____/ \____/
8  * M[0]--|       |       |--M[2]
9  *         M[1]-|
10 */
```

Matrixaufgabe

Allokiert eindimensionales Array dynamisch vom Heap

```
1 int (*M)[6] = malloc(sizeof (*M[0]) * sizeof(*M) * 3);
2 /**
3  * Bild im Speicher:
4  * p:   [iiiiii|iiiiii|iiiiii]
5  *       \____/ \____/ |-- (*(M+2))[0]
6  *       /^     |
7  * M: (p)       |-- *(M+1)
8  *
9  * Anstatt Pointer-Derefenzierung mit Array-Notation:
10 * M[1] = *(M+1)
11 * M[2][0] = (*(M+2))[0]
12 */
```

Matrixaufgabe

Allokiert Speicher von gleicher Größe wie das Array dynamisch

```
1  int *M = malloc(sizeof (*M) * 6 *3);
2  /**
3   * Bild im Speicher:
4   * p:    [iiiiii iiiiii iiiiii]
5   *       /^       |
6   * M: (p)         |- *(M+6) = *(M+ 1*6 +0)
7   *
8   * Anstatt Pointer-Derefenzierung mit Array-Notation:
9   * M[6] = *(M+6)
10  * M[1*6 +0] = *(M+ 1*6 +0)
11  */
```

Matrixaufgabe

Allokiert Speicher für weitere Zeiger die auf einzelne Zeilen zeigen

```
1  int **M = malloc(sizeof (*M) *3);
2  for (size_t i = 0; i < 3; i++) { *(M+i) = malloc(sizeof (**M) * 6); }
3  /**
4   * Bild im Speicher:
5   * p: [iiiiiii]     p: [iiiiiii]     p: [iiiiiii]
6   *      ^\ ____________/^               /^  |
7          | | ____________________________|   |-- *(*(M+2)+3)
8          | | |
9   * p:   [p p p]
10  *      /^   |
11  * M: (p)    |-- *(M+2)
12  *
13  * Anstatt Pointer-Dereferenzierung mit Array-Notation
14  * M[2] = *(M+2)
15  * M[2][3] = *(*(M+2)+3)
16  */
```

Pointer aufs Nix

Es gibt einen „untypisierten“ Pointer: `void*`

- Zeigt auf keinen (konkreten) Typen
- Somit auch keine wohldefinierte Pointer-Arithmethik möglich!
- Funktionen wie `malloc()` geben `void*` zurück, da der eigentliche Typ nicht bekannt ist
- Werden automatisch bei Zuweisungen o. Ä. in einen konkreten Typ umgewandelt („gecastet“):

```
1 int *M[42] = malloc(sizeof (*M) * sizeof (*M[0]));
```

Terminals

Physische Terminals

- Das (Text-)Interface zu einem Rechner nennt man **console**
- Um auf dieses zuzugreifen, benötigt es Tastatur & Anzeige. Dies bildet einen Endpunkt, sog. **Terminal**
- Früher waren das **TeleTYprewriter**, kurz **TTY**
- Diese waren physisch direkt seriell angeschlossen

Terminals

Virtuelle Terminals & Pseudo-Terminals

- Eine Abstraktionsebene darüber gibt es **Virtual Terminals**, kurz **VT**
- Diese laufen in der Software, Zugriff in Linux mit **[Ctrl]**+**[Alt]**+**[F1–F6]**
- Insbesondere kann auch der ***X*-Server** (für Grafik) darauf laufen
- **Pseudo-Terminals** sind ähnlich zu VTs, können aber ad-hoc erstellt werden

Terminals

Terminal Emulatoren

- Auf einer grafischen Oberfläche möchte man oft auch auf die Konsole zugreifen
- Abhilfe schaffen **Terminal Emulatoren**, das ist das was ihr in aller Regel benutzt habt!
- Diese verbinden sich auf je ein Pseudo-Terminal

Shells

Kommandozeilen-Interpreter

- Alles ist Shell – ihr könnt euren Terminal Emulator dazu bringen, eure Programme als Shells auszuführen!
- … aber das ist selten sinnvoll, oft werden Bash oder Zsh genutzt.
- Eine Shell ist das, was eure Eingaben nimmt und Programmaufrufe ausführt

Einstellungen für's Terminal

Wir haben viele unterschiedliche Arten von „Terminals“

- Müssen herausfinden welche Optionen diese Unterstützen ...
  - termcap (1978) und terminfo (1981/82)
- ... und ggf. etwas umkonfigurieren
  - ioctl (UNIX system call) und termios-API (POSIX Standard)
  - Bspw. in den RAW (non-canonical) Modus schalten für direkten Input (vi, nano, ... oder euer Konsolenspiel?)
- Einiges davon wird automatisch von Bibliotheken wie (n)curses oder termbox übernommen (Letzteres werden wie benutzen)

<stdio.h>-Header

C-Funktionen zum Öffen/Schließen & Lesen/Schreiben

```c
// FILE *fopen(const char *restrict pathname, const char *restrict mode);
FILE *my_file = fopen("path/to/file.txt", "rw");

// size_t fread(void *restrict ptr, size_t size, size_t nitems,
//         FILE *restrict stream);
char buf[64]; // sizeof buf[64] == 64, da Längeninformation vorhanden!
size_t n = fread(buf, sizeof buf[0], sizeof buf, my_file);
printf("read: %64s\n", buf);
scanf("%63s", buf); buf[63] = '\0';

// size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
//         FILE *restrict stream);
fwrite(buf, sizeof buf[0], strlen buf, my_file);

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

<stdio.h>-Header

Einige weitere Funktionen:

- `fgetc()` und `fputc()`
- `fgets()` und `fputs()`
- `ungetc()`
- `fseek()`/`fsetpos()` und `ftell()`/`fgetpos()`

Drei `FILE*` sind schon vordefiniert: `stdin`, `stdout` und `stderr`.
Achtung: Statt `gets()` lieber `fgets()` mit `stdin` als Parameter nehmen!

<math.h>-Header

Funktionen für komplexere Mathematik:

- Trigonometrische Funktionen `sin()`, `cos()`, `asin()`, `acos()`, …
- Exponentialfunktionen `exp()`, `exp2`, …
- …

Jedoch: Ist zwar teil der Standardbibliothek, wird jedoch per Default nicht in euer Programm „miteingebaut“ (**gelinkt**).

Linking

Bisher: Eine C-Datei entspricht einem Programm.

- Aber eigentlich: Eine Kompilationseinheit (i. d. R eine C-Datei) entspricht einer Objektdatei:

```
1 $ cc -std=c11 -Wall -Wextra -pedantic -o a.o a.c -c  # erstellt a.o
```

- Eine Objektdatei enthält den reinen kompilierten (übersetzten) Programmcode (in Maschinensprache)
- Noch nicht ausführbar, müssen noch gelinkt werden:

```
1 $ cc -o a a.o
```

- „Heimlich“ werden hier jedoch noch weitere Bibliotheken einfach dazu gelinkt, bspw. die C-Standardbibliothek (dynamisch) und die C-Runtime (statisch)

Linking

- Um mehrere Module zusammenzulinken:

```
1 $ cc -o programm a.o b.o c.o
```

- Um eine Bibliothek (Library) hinzuzulinken, hier die Mathe-Bibliothek (m):

```
1 $ cc -o programm a.o -lm
```

- Kann auch alles in einem Schritt gemacht werden:

```
1 $ cc -o programm a.c b.c c.c -lm
```

- Manchmal ist es aber sinvoll dies einzeln zu machen – um das zu automatisieren: Makefiles (s. Vortrag in 2 Vorlesungen)

Linking – Theorie

- Modul A definiert Funktion $f()$
- Diese wird von Modul B benutzt
- Die Deklaration der Funktion f im Header zu A dient nur der Überprüfung der Übergabe der konkreten Parameter etc.
- Erst bei dem Linken wird der Verweis aufgelöst
  - Dazu gibt es in den Modulen Informationen über die (extern) definierten Funktionen & Variablen als auch über unaufgelöste Verweise
  - Ist das Auflösen nicht möglich, gibt es Fehler:

```
1 $ cc -o programm B.o
2 B.o: In function `main':
3 B.c:(.text+0x1e): undefined reference to `f'
```

Weiterführende Quellen I

- The Open Group
  POSIX: <termios.h>
  http://pubs.opengroup.org/onlinepubs/9699919799/
- Diverse Autoren.
  Wikibooks
  https://en.wikibooks.org/wiki/C_Programming
- Jens Gustedt.
  Modern C
  http://icube-icps.unistra.fr/index.php/File:ModernC.pdf