

Mehr zu Speicher

Ausblick

Heute lernen wir ...

- ▶ was der Präprozessor ist und wie man ihn nutzt
- ▶ wie wir dynamisch Speicher allozieren können
- ▶ ein paar fortgeschrittene Zeiger-Tricks
- ▶ wie man mit mehr als einer Datei arbeitet
- ▶ wie man Dateien öffnet, ließt, und schreibt

Wiederholung – Zeiger

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

dynamische Speicherallokation

- ▶ Was, wenn wir nicht vorher wissen, wie viel Speicher wir brauchen?
- ▶ dynamische Speicherallokation mit `malloc()` aus `<stdlib.h>`
- ▶ der Speicher wird mit `free()` wieder frei gegeben
- ▶ der `sizeof`-Operator ist ein Präfixoperator der die Größe eines Objektes bestimmt
- ▶ ideal zur Bestimmung der benötigten Speichergröße für `malloc()`
- ▶ für Arrays muss die Speichergröße mit der Anzahl der Elemente multipliziert werden

dynamische Speicherallokation

```
1 int *foo = malloc(sizeof *foo); // alloziert einen int
2 if (foo == NULL)                // Anforderung erfolgreich?
3     perror("malloc");           // Fehlerbehandlung!
4 *foo = 1337;
5 free(foo);                       // Speicher freigeben
6
7 foo = malloc(16 * sizeof *foo); // alloziere 16 int
8 if (foo == NULL) /* ... */ ;
9 foo[14] = 4711;
10 free(foo);
```

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.h. 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)
```

Matrizen

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 */
```

Matrizen

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-Dereferenzierung mit Array-Notation:
10 * M[1] = *(M+1)
11 * M[2][0] = (*(M+2))[0]
12 */
```


Matrizen

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-Dereferenzierung mit Array-Notation:
9  * M[6] = *(M+6)
10 * M[1*6 +0] = *(M+ 1*6 +0)
11 */
```

Matrizen

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: [iiiiii]      p: [iiiiii]      p: [iiiiii]
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-Arithmetik 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]));
```

Der Präprozessor

- ▶ Der Präprozessor läuft vor dem Compiler und bereitet den Quelltext für den Compiler auf
- ▶ Die Arbeit des Präprozessors kann mit `gcc -E quelltext.c` verfolgt werden
- ▶ Aufgaben: Kommentare entfernen, Direktiven auswerten, Makros ersetzen
- ▶ Das `#` einer Präprozessordirektive muss erstes Zeichen der Zeile sein

Präprozessordirektiven

```
#include "header.h"
```

```
#include <header.h>
```

- ▶ kopiert den Inhalt von `header.h` in das aktuelle Modul
- ▶ `<header.h>` für Systemheader, `"header.h"` für eigene Header

Präprozessordirektiven

```
#define MACRO definition ...  
#define MACRO(x,y,z) definition ...  
#undef MACRO
```

- ▶ Definiert ein Makro. Das Makro wird überall durch seine Definition ersetzt
- ▶ Funktionsartige Makros haben Argumente, die mit ersetzt werden
- ▶ Vorsicht bei der Klammerung in funktionsartigen Makros
- ▶ Makros können mit `#undef` aufgehoben werden

Präprozessordirektiven

```
#ifdef MACRO  
#if ausdruck  
#else  
#elif ausdruck  
#endif
```

- ▶ Wie if-Anweisung, wird aber im Präprozessor ausgeführt
- ▶ spezieller Operator `defined()` prüft, ob Makro definiert ist
- ▶ `elif` ist wie `else if`
- ▶ nützlich, um das Programm beim kompilieren anpassen zu können

Präprozessordirektiven

```
1 #if defined(__linux)
2 # define OS "Linux"
3 #elif defined(__FreeBSD__)
4 # define OS "FreeBSD"
5 #elif defined(_WIN32) || defined(_WIN64)
6 # define OS "Windows"
7 #else
8 # error Unbekanntes Betriebssystem!
9 #endif
```


Präprozessordirektiven

```
#error Nachricht ...
```

```
#pragma ...
```

```
#line 12 "foo.c"
```

```
#
```

- ▶ `#error` bricht die Kompilation mit Fehlermeldung ab
- ▶ `#pragma` stellt kompilerspezifische Erweiterungen bereit, z. B. `#pragma omp` für OpenMP oder `#pragma STDC` für spezielle C-Features
- ▶ `#line` setzt Zeilennummer und Dateiname für Warnungen und Fehlermeldungen
- ▶ `#` ist die leere Direktive und macht nichts

Der <stdio.h>-Header

C-Funktionen zum Öffnen/Schließen & Lesen/Schreiben

```
1 // FILE *fopen(const char *restrict pathname, const char *restrict mode);
2 FILE *my_file = fopen("path/to/file.txt", "w+");
3
4 // size_t fread(void *restrict ptr, size_t size, size_t nitems,
5 //             FILE *restrict stream);
6 char buf[64]; // sizeof buf[64] == 64, da Längeninformation vorhanden!
7 size_t n = fread(buf, sizeof buf[0], 64, my_file);
8 printf("read: %64s\n", buf);
9 scanf("%63s", buf); buf[63] = '\\0';
10
11 // size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
12 //             FILE *restrict stream);
13 fwrite(buf, sizeof buf[0], strlen(buf), my_file);
14
15 // int fclose(FILE *stream);
16 fclose(my_file);
```

Der `<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!

Der `<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**). Zur Benutzung `-lm` als letzten Operand an den Compiler übergeben.

Linken

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)

Linken

- 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 sinnvoll dies einzeln zu machen – um das zu automatisieren: Makefiles (s. Vortrag in 2 Vorlesungen)

Linken – 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
2B.o: In function `main':
3B.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>