Die libc und mehr

Ausblick

Heute lernen wir ...

- wie man C-Programme aus mehreren Dateien baut
- wie man Header-Dateien schreibt
- was es mit *Speicherklassen*, *Typqualifizierern*, und *Verlinkung* auf sich hat
- und was sich so alles an Funktionen in der libc anfindet

Arbeiten mit mehreren Modulen

- Bisher: Eine Programm besteht aus einer C-Datei
  - … und den Headern der C-Standardbibliothek (`<stdio.h>`, `<stdlib.h>`, …)
  - … und der Bibliothek selber (`crt0.o`, `libc.so`)
- Bei größeren Projekten bietet es sich an, das Programm auf mehrere Kompilationseinheiten bzw. Module aufzuteilen
- Nach welchen Kriterien aufgeteilt wird, hängt vom Projekt ab. Damit beschäftigt sich u. A. die Softwaretechnik (Entwurfsmuster)

Beispiel

a.c

```c
static int bar(int a) {
    return (a/2);
}
int foo(int a, int b) {
    return (bar(a)+b+42);
}
```

b.c

```c
#include <stdio.h>
extern int foo(int a, int b);
int main(void) {
    printf("%d\n", foo(1, 2));
}
```

Beispiel

Kompilation ohne Linken:

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

Wir können uns alle globalen Symbole anzeigen lassen:

```
$ nm -g a.o
0000000000000015 T foo
$ nm -g b.o
                 U foo
0000000000000000 T main
                 U printf
```

Beispiel

Manuelles Linken der einzelnen Dateien danach:

```
$ cc -o prog a.o b.o
```

Und nun eine (im Beispiel unvollständige) Liste der globalen Symbole:

```
$ nm -g prog
000000000000114e T foo
0000000000001172 T main
                 U printf@@GLIBC_2.2.5
0000000000001040 T _start
```

Beobachtungen

- Die undefinierte Referenz in `b.o` auf `foo` aus `a.o` wurde aufgelöst
- Es kam ein neues Symbol `_start` hinzu
  - Ist unter Linux der eigentliche Programmeintrittspunkt, ruft u. A. `main()` auf!
  - Gehört zur C-Runtime (oft `crt0.o`/`crt1.o`), nötig für die Benutzung einiger Funktionen der libC
  - Wird implizit statisch hinzugelinkt
- Die Referenz auf `printf` wurde ersetzt durch `printf@@GLIBC_2.2.5`
  - Ist in der `libc.so` definiert (hier GNU LibC)
  - Wird implizit *dynamisch* hinzugelinkt (`-lc`)

Zwischen-Zusammenfassung

- Funktionen oder Variablen (Symbole!), auf die von anderen Übersetzungseinheiten zugegriffen werden sollen, müssen global sein!
  - Somit natürlich auch `main()` (wird von `_start` aufgerufen)
- Wird von einer anderen Einheit auf ein Symbol außerhalb zugegriffen, sollte dieses als `extern` deklariert werden
  - … Header-Dateien sind Dateien die (vor Allem) solche externen Deklarationen enthalten
  - Deklarationen ersetzen nicht das Linken, also das Bereitstellen der Definitionen – lediglich Hilfestellung für den Compiler!
- Nur interne Symbole, welche außerhalb aller Funktionen („Top-Level“) definiert werden, sollten besser als `static` markiert werden

Programmaufbau

- Für jedes Modul wird eine C-Datei angelegt
- Für jedes globale Symbol darin eine externe Deklaration in eine Header-Datei, oft gleichen Namens
- Benutzt man Funktionalität aus *A* in einem anderen Modul *B*, wird diese per `#include "A.h"` eingebunden
  - Werden Teile aus *A* bereits in dem Header von *B* benötigt, wird bereits dort eingebunden
  - … bspw. Typen-Definitionen wie `size_t`
- Achtung: Doppeltes Einbinden

Include-Guards

- Wird ein Header mehrmals in dem gleichen eingebunden, kann das Probleme bereiten
- Doppelte reine Deklarationen sind egal
- Aber Definitionen, bspw. von Variablen, Typen oder Konstanten nicht!
- Ein einfacher Schutz sind Include-Guards in jedem Header

```
1 #ifndef MODULNAME_H
2 #define MODULNAME_H
3
4 #define BUFFER_SIZE 1024
5 extern const int foo;
6 struct bar { int baz; }
7
8 #endif // MODULNAME_H
```

Deklarationen und Definitionen

Wir müssen Deklarationen und Definitionen unterscheiden:

- Eine *Deklaration* macht einen Bezeichner dem Kompiler bekannt. Die selbe Variable oder Funktion kann beliebig oft deklariert werden
- Eine *Definition* erzeugt gleichzeitig eine Variable oder definiert eine Funktion. Jede Variable oder Funktion muss genau einmal definiert werden; jede Definition ist zugleich eine Deklaration

$\longrightarrow$ Jede globale Variable oder Funktion muss genau in einer Datei definiert sein und darf sonst nur deklariert werden (*one definition rule*).

Speicherklassen

Speicherklassen bestimmen, wie lange ein Objekt am Leben ist.

- Ein *automatisches* Objekt wird alloziert und dealloziert, wenn der es umgebende Block betreten bzw. verlassen wird (wie Stackvariable in ASM)
- Ein *statisches* Objekt wird beim Programmstart alloziert und beim Programmende dealloziert (wie *DD* Variable in ASM)
- Ein *dynamisches* Objekt wird mit `malloc()` alloziert und mit `free()` dealloziert
- Ein *Thread-lokales* Objekt ist wie ein statisches Objekt, aber jeder Thread hat seine eigene Kopie. Hier nicht weiter betrachtet.

Verlinkung

Besteht ein Programm aus mehreren Modulen, so bestimmt die *Verlinkung* eines Bezeichners, wie dieser über Modulgrenzen gehandhabt wird.

- Alle Bezeichner gleichen Namens mit *externer Verlinkung* bezeichnen das selbe Ding innerhalb eines Programmes, wie *global* in ASM
- Alle Bezeichner gleichen Namens mit *interner Verlinkung* bezeichnen das selbe Ding innerhalb eines Moduls
- Alle Bezeichner gleichen Namens *ohne Verlinkung* bezeichnen verschiedene Dinge

Speicherklassenspezifizierer

Speicherklasse und Verlinkung eines Bezeichners kann durch einen *Speicherklassenspezifizierer* angegeben werden. Dieser wird in einer Deklaration dem Typ vorangestellt.

`static`	bestimmt statische Speicherklasse und interne Verlinkung
`extern`	bestimmt statische Speicherklasse und externe Verlinkung (Standard für globale Variablen). Unterscheidet zwischen Variablendefinition und -deklaration
`auto`	bestimmt automatische Speicherklasse ohne Verlinkung, Standard für lokale Variablen, fast nie explizit angegeben
`register`	wie `auto` mit Hinweis, Variable in Register zu halten, obsolet
`_Thread_local`	bestimmt Thread-lokale Speicherklasse und externe Verlinkung
`typedef`	erzeugt einen Typ-Alias anstatt einer Variable

Deklarationen und Definitionen

```
1  int a;          // def. statische Variable, externe Verl.
2  extern int a;   // dekl. statische Variable, externe Verl.
3  static int a;   // def. statische Variable, interne Verl.
4
5  int foo(void);            // dekl. Funktion, externe Verl.
6  extern int foo(void);     // dito
7  static int foo(void);     // dekl. Funktion, interne Verl.
8
9  int foo(void) { }         // def. Funktion, externe Verl.
10 extern int foo(void) {}   // dito
11 static int foo(void) {}   // def. Funktion, interne Verl.
```

Typqualifizierer

Typqualifizierer verändern Typen. Sie werden je nachdem, was sie qualifizieren, an verschiedene Stellen geschrieben.

`const`	Variable/Objekt darf ggf. nach Initialisierung nicht beschrieben werden
`restrict`	auf Objekt wird nur durch diesen Zeiger zugegriffen
`volatile`	der Kompiler darf Zugriffe auf das Objekt nicht optimieren
`_Atomic`	Variable oder Objekt ist atomar
`_Noreturn`	Funktion kehrt nicht zurück

Typqualifizierer

```
1 const int x;              // Konstanter int
2 const int *x;             // Zeiger auf konstanten int
3 int const *x;             // dito
4 int *const x;             // Konstanter Zeiger auf int
5 int *restrict x;          // restrict-Zeiger auf int
6 int x[restrict 12];       // restrict-Array von 12 int
```

Die libc – assert.h

Der Header <assert.h>

```c
#include <assert.h>

void assert(int expr);
```

- Prüft, ob der übergebene Ausdruck wahr ist (Zusicherung). Falls nicht, wird das Programm abgebrochen.
- Sinnvoll, um Invarianten zu prüfen
- Ist `NDEBUG` definiert, wird `assert` ignoriert

Die libc – ctype.h

`isalpha()`, `isalnum()` ist Zeichen Buchstabe oder Buchstabe/Ziffer?

`isdigit()`, `isxdigit()` ist Zeichen Ziffer oder hexadezimale Ziffer?

`isupper()`, `islower()` ist Zeichen Groß- oder Kleinbuchstabe?

`isblank()`, `isspace()` ist Zeichen Weißraum oder Leerzeichen/Tab?

`iscntrl()`, `ispunct()` ist Zeichen Steuer- oder Satzzeichen?

`isprint()`, `isgraph()` ist Zeichen druckbar (mit/ohne Leerzeichen)?

`toupper()`, `tolower()` wandle Zeichen zu Groß/Kleinbuchstaben

Die libc – errno.h

```c
#include <errno.h>

extern int errno;
```

- `errno` enthält die Nummer des letzten Fehlers
- Fehlercodes sind im Handbuch dokumentiert
- die symbolischen Konstanten wie `EINVAL`, `EIO`, `EPERM`, … sind auch hier definiert
- Fehlernachricht kann mit `strerror()` bestimmt und mit `perror()` ausgegeben werden (aus `<string.h>`, `<stdio.h>`)

Die libc – limits.h

`CHAR_BIT`	Anzahl der Bits pro Byte (acht)
`type``_MIN`	kleinster Wert eines vorzeichenbehafteten Typs (`CHAR`, `SHRT`, `INT`, `LONG`, `LLONG`)
`type``_MAX`	größter Wert eines vorzeichenbehafteten Typs
`U`*`type`*`_MAX`	größter Wert eines vorzeichenlosen Typs

$\longrightarrow$ weitere Limits in `<stdint.h>`.

Die libc – math.h

`sin()`, `cos()`, `tan()` Winkelfunktionen

`asin()`, `acos()`, `atan()`, `atan2()` inverse Winkelfunktionen

`exp()`, `pow()`, `log()` Exponential-, Potenz-, und Logarithmusfunktionen

`sqrt()`, `cbrt()` Quadrat- und Kubikwurzeln

`fabs()`, `floor()`, `ceil()` Betrag, auf- und abrunden

`fdim()`, `fmax()`, `fmin()` Betrag der Differenz, Maximum, Minimum

$\longrightarrow$ und viele mehr

Die libc – setjmp.h

```
1 #include <setjmp.h>
2
3 int setjmp(jmp_buf env);
4 _Noreturn void longjmp(jmp_buf env, int val);
```

- mit `setjmp()` kann man einen Sprung setzen
- und `longjmp()` springt zum gesetzten Sprung
- gibt `setjmp()` den Wert null zurück, ist es von selbst zurückgekehrt
- sonst als Konsequenz eines Sprungs mit `longjmp()`
- gesetzter Sprung kann von überall aus angesprungen werden, solange Funktion in der der Sprung gesetzt wurde aktiv ist

Die libc – setjmp.h

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

int main(void) {
  jmp_buf env; int i;

  if (i = setjmp(env), i != 0)
    printf("wiedergekommen mit i = %d\n", i);
  else {
    printf("i = %d, springe ...\n", i);
    longjmp(env, 42);
  }
}
```

Die libc – stdarg.h

```
1 #include <stdarg.h>
2
3 type va_start(va_list ap, type);
4 type va_arg(va_list ap, type);
5 void va_end(va_list ap);
6 void va_copy(va_list dest, va_list src);
```

Die libc – stdarg.h

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

void echo(const char *str0, ...) {
  va_list ap; char *str;

  printf("%s", str0);
  va_start(ap, str0); /* letzter Parameter vor ... */
  while (str = va_arg(ap, char *), str != NULL)
    printf(" %s", str);
  putchar('\n');
  va_end(ap);
}
```

Die libc – stdbool.h

```
1 #include <stdbool.h>
2
3 #define bool _Bool
4 #define true 1
5 #define false 0
```

Die libc – stdio.h

`remove()`, `rename()` Dateien löschen und umbenennen

`fopen()`, `freopen()`, `fclose()` Dateien öffnen und schließen

`stdin`, `stdout`, `stderr` `FILE*` für Standardeingabe, -ausgabe, -fehler

`setvbuf()`, `fflush()` Pufferung konfigurieren, Puffer rausschreiben

`fgetc()`, `fputc()`, `ungetc()` Zeichen lesen, schreiben, unlesen

`fread()`, `fwrite()` Binäredaten lesen, schreiben

`fgets()`, `fputs()` Strings lesen, schreiben

`fseek()`, `ftell()` Position bestimmen, spulen

`ferror()`, `feof()`, `clearerr()` Fehler- und Dateiendestatus auslesen, löschen

`fscanf()`, `fprintf()` formatierte Ein- und Ausgabe

$\longrightarrow$ und viele mehr

Die libc – stdlib.h

`strtol()`, `strtod()` Zeichenketten in Ganz- und Gleitkommazahlen parsen

`srand()`, `rand()` Zufallszahlengenerator initialisieren, benutzen

`calloc()`, `realloc()` Arrays allozieren, Objekte reallozieren

`abort()`, `exit()`, `system()` Programm abbrechen, beenden, Befehle ausführen

`getenv()`, `putenv` Umgebungsvariablen auslesen, setzen

`bsearch()`, `qsort()` Arrays durchsuchen, sortieren

$\longrightarrow$ und viele mehr

Die libc – string.h

`memcpy()`, `strcpy()`, `strcat()` Arrays, Strings kopieren, konkatenieren

`memcmp()`, `strcmp()` Arrays, Strings lexikographisch vergleichen

`memchr()`, `strchr()`, `strstr()` in Arrays/Strings nach Zeichen/strings suchen

`strspn()`, `strcspn()` nach Zeichen suchen, die in Menge nicht vorkommen / vorkommen

`strsep()`, `strpbrk()`, `strtok` Strings an Separatoren zerhacken

`strlen()`, `memset()` String-Länge bestimmen, Arrays initialisieren

$\longrightarrow$ und viele mehr

Die libc – string.h

```c
#include <stdlib.h>
#include <string.h>

static int compar(const void *a, const void *b) {
  return strcmp(*(const char **)a, *(const char **)b);
}

/* ein Array von Strings sortieren */
void strsort(char **array, size_t n) {
  qsort(array, n, sizeof *array, compar);
}
```

Die libc – time.h

`time()` aktuelle Uhrzeit in Sekunden seit 1. Januar 1970 bestimmen

`clock()` Tick-Zähler auslesen, tickt pro Sekunde `CLOCKS_PER_SEC` mal (aber nur wenn das Programm läuft, also nicht, wenn es wartet)

`ctime()` Zeit als String darstellen

$\longrightarrow$ und weitere

Die libc – time.h

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

int main(void) {
  clock_t start, end;
  time_t jetzt = time(NULL);
  puts(ctime(&jetzt));
  start = clock();
  for (volatile int i = 0; i < 1000000000; i++);
  end = clock();
  printf("Es sind %f Sekunden vergangen.\n",
      (double)(end - start) / CLOCKS_PER_SEC);
}
```