C Projekte aus mehreren Modulen

Ziel

- 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 const int foo = 42;
6 struct bar { int baz; }
7
8 #endif // MODULNAME_H
```