6.8 Extras/Modularisierung in C

6.8.1 Wiederverwendung von Funktionen an einem Beispiel

Häufig kommt es vor, dass man bei der Erstellung eines Programmes auf eine Aufgabe stößt, die man selbst (oder jemand anders aus der Firma) schon an anderer Stelle gelöst hat. Oft handelt es sich um eine Funktion, die in einem anderen Programm eingesetzt wurde. Wie kann man es nun hinbekommen, dass eine Funktion in mehreren Programmen eingebaut werden kann?

Als Beispiel soll ein Programm meine wunderbare existierende Funktion `scandouble` benutzen:

uri.c

```c
#include <stdio.h>

int main(void)
{
    double u, r, i;
    u=scandouble("Spannung in V eingeben ....: ");
    r=scandouble("Widerstand in Ohm eingeben: ");
    i=u/r;
    printf("Stromstaerke in A.........: %g\n", i);
    return 0;
}
```

Bei `uri.c` fehlt jetzt noch die Definition von `scandouble`. Natürlich könnte man die Funktion aus dem alten Programm herauskopieren und hier einfügen (so genanntes *copy and paste*). Das wirkt jedoch sehr unprofessionell und hat auch erhebliche Nachteile. Wenn man im alten Programm an `scandouble` eine Änderung anbringt (z. B. die Korrektur eines Fehler), dann gilt die Änderung nur dort. Wenn man nicht weiß, wo man selbst oder jemand anders eine Kopie dieser Funktion hat, bleibt die Kopie von der Änderung ausgeschlossen.

Daher ist es besser, wenn die Funktion aus dem alten Programm in eine eigene Datei gebracht wird:

myscan.c

```c
#include <stdio.h>
double scandouble(const char *aufforderung)
{
    int rc;
    double erg=0.0;
    do{
        printf("%s", aufforderung);
        rc=scanf("%lf", &erg);
        while(getchar()!='\n'){}
    } while(rc<=0);
    return erg;
}
```

Jetzt gäbe es wieder eine einfachere Möglichkeit, die Funktion in unser Programm zu bringen:

uri.c

```c
#include <stdio.h>
#include "myscan.c"
int main(void)
...
```

Das ist nicht so schlimm wie die Methode des *copy and paste*, und es funktioniert sogar gut:

Terminal
```
schueler@debian964:~$ gcc uri.c
schueler@debian964:~$ ./a.out
...
```

6.8.2 Compilieren und Linken

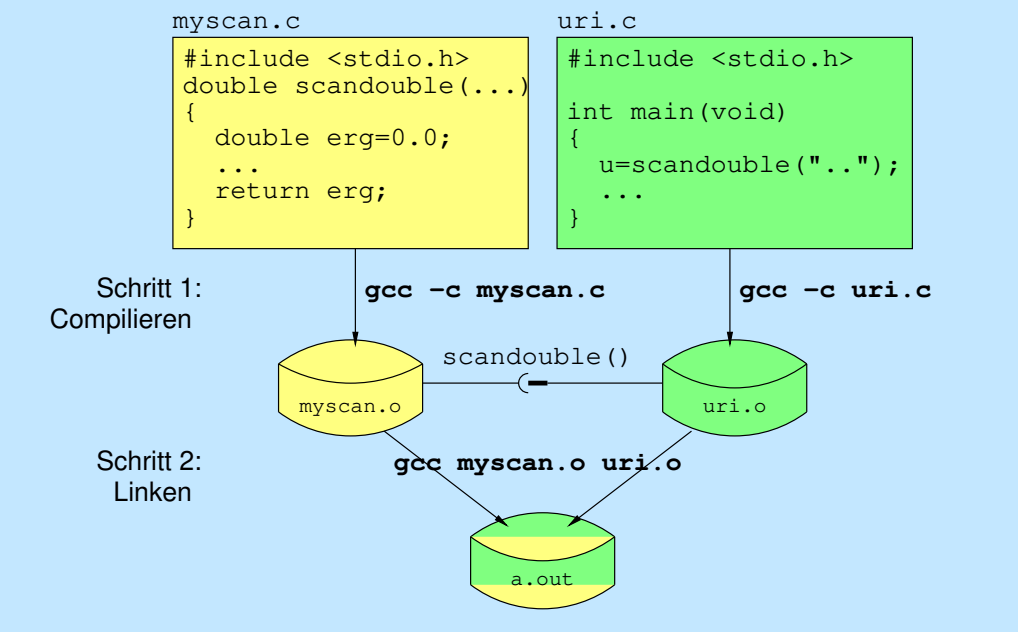

Abbildung 1: Compilieren und Linken

Die in der Sprache bevorzugte Methode dagegen geht einen anderen Weg (Abbildung 1):

a) Im ersten Schritt wird jede Datei (`uri.c` und `myscan.c`) für sich (also einzeln) compiliert.

b) Im zweiten Schritt werden die beiden Objektdateien zusammen mit der Standard-Bibliothek und dem Startcode zu einem Programm zusammengebunden; man sagt, sie werden *gelinkt*.

Der erste Schritt am Beispiel von `myscan.c`:

Terminal
```
schueler@debian964:~$ gcc -c myscan.c
schueler@debian964:~$ ls myscan.*
myscan.c myscan.o
```

Mit der Option `-c` bekommt der Compiler gesagt, dass er nur compilieren soll und noch nicht linken. Jetzt der erste Schritt am Beispiel von `uri.c`:

Terminal
```
schueler@debian964:~$ gcc -c uri.c
gcc -c uri.c
uri.c: In function 'main':
uri.c:5:6: warning: implicit declaration of function 'scandouble'
    [-Wimplicit-function-declaration]
    u=scandouble("Spannung in V eingeben....:");
schueler@debian964:~$ ls uri.*
uri.c uri.o
```

Die Warnung sagt aus, dass für die Funktion `scandouble` an der Stelle, an der sie aufgerufen wird, keine Funktionsdefinition, aber auch kein Prototyp bekannt ist. Dieses Problem soll weiter unten gelöst werden.

Nun zum zweiten Schritt: Die beiden Objektdateien sind noch nicht ausführbar. Eine enthält zwar `main`, aber es fehlt eine Funktion, die andere enthält die fehlende Funktion, aber keine `main`-Funktion. Zusammengebunden werden die beiden Dateien über das Linker-Programm `ld` (bei Linux) oder `link.exe` (bei Windows):

Terminal

```
schueler@debian964:~$ ld uri.o myscan.o
ld: warning: cannot find entry symbol _start; defaulting to
00000000004000b0
myscan.o: In function 'scandouble':
myscan.c:(.text+0x29): undefined reference to 'printf'
myscan.c:(.text+0x41): undefined reference to '__isoc99_scanf'
myscan.c:(.text+0x4a): undefined reference to 'getchar'
uri.o: In function 'main':
uri.c:(.text+0x6d): undefined reference to 'printf'
```

Hier sind zwei Fehler versteckt: Erstens fehlt der Startcode, so dass das Programm nicht gestartet werden könnten. Zweitens fehlt die C-Standard-Bibliothek mit den nützlichen Funktionen `printf`, `scanf` und `getchar`. Also muss man den Linker nur noch mit den richtigen Optionen versehen, und alles läuft. Diese Aufgabe kann unser Compiler-Programm `gcc` übernehmen. Es compiliert nicht nur, sondern ruft den Linker so auf, dass der Startcode und die C-Standard-Bibliothek[1] jedes Mal hinzugelinkt werden:

Terminal

```
schueler@debian964:~$ gcc uri.o myscan.o
schueler@debian964:~$ ls a.out
a.out
schueler@debian964:~$ ./a.out
Spannung in V eingeben....: 20
Widerstand in Ohm eingeben: 4
Stromstärke in A.........: 1
```

6.8.3 Header-Dateien

Das Programm läuft, es gibt jedoch erstaunliche Ergebnisse aus. Das erinnert (nach einigem Nachdenken) an die Warnung des Compilers, dass an der Stelle des Aufrufs kein Prototyp für die Funktion `scandouble` bekannt ist.

Das ist ein Problem, wenn man ein Programm aus mehreren Teilen zusammenbauen will. Der Linker setzt alles zusammen, was den gleichen Namen trägt[2]. Aber der Compiler möchte vorher gerne wissen, wie viele Parameter die Funktion hat und welche Typen diese Parameter haben. Außerdem braucht er den Rückgabetyp. Ohne diese Informationen kann er bei den Parametern keine Typprüfung durchführen (vielleicht nicht schlimm, wenn der Programmierer keine Fehler macht). Vor allem aber nimmt er bei fehlenden Informationen an, dass der Rückgabetyp `int` ist!

Diese Informationen sind jedoch in der Objektdatei `myscan.o` so nicht mehr vorhanden[3]. An dieser Stelle kommt die *Header-Datei* ins Spiel: Man nimmt einfach die Prototypen aller Funktion aus `myscan.c` und schreibt sie in die Datei `myscan.h`:

myscan.h

```
1 double scandouble(const char *aufforderung);
```

[1] außer den Funktionen aus `<math.h>`

[2] Bei C++ ist das anders: Da bekommt der Linker auch eine Typ-Information für die Parameter.

[3] Das könnte man natürlich ändern, indem man das Linkerformat ändert.

Diese (und *nur* diese!) Header-Datei wird jetzt in `uri.c` mit dem Befehl `#include` eingebunden[4]:

uri.c

```
1 #include <stdio.h>
2 #include "myscan.h"
3 int main(void)
4 ...
```

Jetzt kann man `uri.c` ohne Warnung compilieren:

Terminal

```
schueler@debian964:~$ gcc -c uri.c
schueler@debian964:~$ gcc uri.o myscan.o
```

Viel wichtiger: Der Prototyp stellt nun sicher, dass die übergebenen Daten und vor allem der Rückgabetyp richtig interpretiert werden:

Terminal

```
schueler@debian964:~$ ./a.out
Spannung in V eingeben....: 20
Widerstand in Ohm eingeben: 4
Stromstärke in A.........: 5
```

6.8.4 Compilieren und Linken in einer Befehlszeile

Das Programm `gcc` kann Quelltext- und Objektdateien am Namen unterscheiden. Deshalb kann `gcc` mit nur einem Befehlsaufruf Programme aus mehreren Quelltextdateien compilieren und linken:

Terminal

```
schueler@debian964:~$ gcc uri.c myscan.c
schueler@debian964:~$ ls a.out
a.out
```

`gcc` kann auch damit umgehen, wenn eine Datei als Quelltext und die andere als Objektdatei vorliegt:

Terminal

```
schueler@debian964:~$ gcc uri.c myscan.o
schueler@debian964:~$ ls a.out
a.out
```

6.8.5 Weitergabe von Software

So kann man seine Funktion (`scandouble`) an andere kostenlos oder gegen Gebühr weitergeben, indem man nur die Objektdatei (`myscan.o`) und die Headerdatei (`myscan.h`) abgibt. Den Quelltext kann man unter Verschluss halten. Auf diese Art sind zahlreiche Bibliotheken für C entstanden, die nahezu alle Anwendungsbereiche abdecken.

[4] Es fällt auf, dass `myscan.h` mit Anführungszeichen statt mit spitzen Klammern geschrieben wird. Das bedeutet, dass es sich um den (relativen oder absoluten) Pfadnamen einer Datei im Dateisystem handelt. Im einfachsten Fall (kein Verzeichnisanteil im Pfadnamen) ist es eine Datei in dem Verzeichnis, das während des Compilierens aktuelles Verzeichnis ist. Beim Einbinden mit spitzen Klammern wird die Datei im System gesucht, meistens in einem speziellen Verzeichnis wie `/usr/include/`.

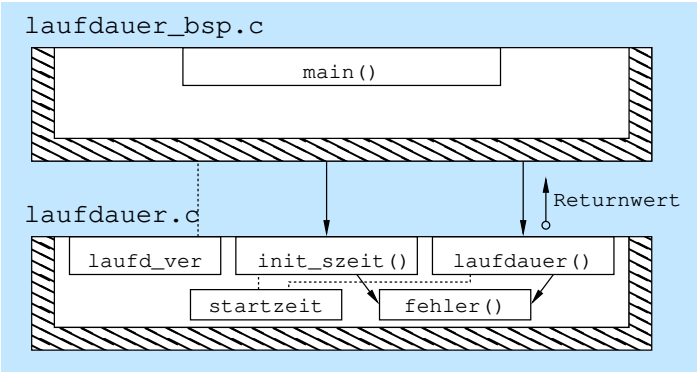

Abbildung 2: Topf-Modell der modularen Programmierung

6.8.6 Eine neue Art zu programmieren: Modulare Programmierung

Ein normales Programm in C und anderen strukturierten Programmiersprachen sieht so aus, dass das Hauptprogramm eine Reihe von Daten (Variablen) besitzt. Diese Daten werden von verschiedenen Funktionen, die sich gegenseitig aufrufen können (in C beliebig), bearbeitet.

Oft will man jedoch einzelne Teile eines Programmes *kapseln*. So soll etwa nur ein Teil eines Programms Zugriff auf bestimmte Hardware haben dürfen; oder nur ein Teil soll mit dem Benutzer Kontakt haben können. Jede dieser Kapseln soll selbständig arbeiten können, eventuell eigene Daten besitzen, die nur von innerhalb der Kapsel verändert werden dürfen. Diese Kapseln nennt man *Module*, der zugehörige Programmierstil heißt *modulare Programmierung*.

Abbildung 2 stellt ein einfaches Beispielprogramm aus zwei solcher Module graphisch dar. Jedes Modul wird als eine Art Topf dargestellt. Die Topfhülle gilt dabei als undurchdringlich. Das obere Modul enthält die Funktion `main`, das untere Modul enthält mehrere Funktionen und Variablen (Funktionen durch runde Klammern gekennzeichnet). Ein Modul kann also Datentypen, Daten und Funktionen enthalten. Die Funktionen in einem Modul können einander aufrufen.

Neu ist: Für jede dieser Komponenten kann der Programmierer des Moduls angeben, ob sie nach außen sichtbar (benutzbar) ist oder nicht.

Ein Modul hat damit zwei Bestandteile, das Interface und die Implementierung.

a) Die öffentlichen, also nach außen sichtbaren Anteile des Moduls heißen ihr *Interface*: im ganzen Programm sichtbare Datentypen, Variablen und Funktionen. Sie werden an der Oberseite des Topfes dargestellt.

b) Der Rest des Moduls ist seine *Implementierung*: private, also nur innerhalb des Moduls benutzte Datentypen, Variablen und Funktionen. Sie werden im Innern des Topfes eingezeichnet.

   Lokale Variablen, Parameter und Rückgabewerte von Funktionen zählen immer zu dieser privaten Kategorie.

Über das Interface greift man von außen auf das Modul zu. Die Implementierung ist das, was die Funktionalität des Moduls ausmacht.

6.8.7 Module in C

In C kann man diesen Programmierstil recht einfach umsetzen. Ein Modul besteht dann aus

a) einer Quelltextdatei (.c)

b) der dazugehörigen Headerdatei (.h)

Die Headerdatei entspricht dann dem Interface, während die Quelltextdatei der Implementierung entspricht. Das passt, weil der Inhalt der Headerdatei für den Compiler der anderen Module sichtbar ist, der Inhalt der Quelltextdatei dagegen nicht.

Hier ist ein Beispiel anhand des Moduls `laufdauer`. Es entspricht dem unteren Modul in Abbildung 2.

laufdauer.c

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

const int laufdauer_ver=100;
time_t startzeit=-1;

void fehler(int nr)
{
    printf("Zeitfehler Nr.%d\n", nr);
    exit(1);
}
void init_szeit(void)
{
    if(startzeit!=-1)
        fehler(1);
    startzeit=time(NULL);
    if(startzeit==-1)
        fehler(2);
}
int laufdauer(void)
{
    return (int)(time(NULL)-startzeit);
}
```

Es gibt dort eine Variable `startzeit`, die zum Programmstart auf die aktuelle Zeit gesetzt werden muss. Dazu dient die Funktion `init_szeit`. Danach kann jederzeit mit der Funktion `laufdauer` die aktuelle Laufzeit des Programms in Sekunden ermittelt werden[5]. Bei Fehlern innerhalb des Moduls wird die Funktion `fehler` aufgerufen, die einen Text ausgibt und das Programm sofort beendet. Mit der Variablen `laufdauer_ver` kann man (wenn man will) die Versionsnummer des Moduls feststellen.

Hier sieht man zwei Vorteile der modularen Programmierung:

a) Man kann eine globale Variable benutzen (oder mehrere davon), und zwar ganz ohne schlechtes Gewissen. Denn diese Variable (`startzeit`) ist nur in dem kleinen Modul sichtbar, nicht aber im Rest des Programms. Es handelt sich also nur um eine modul-globale Variable. Damit entfallen Parameterübergaben und Rückgabewerte.

b) Das Modul ist quasi um die modul-globale Variable herumgebaut worden. Es hat mit dieser Variable ein Gedächtnis bekommen, einen Zustand. Der Zustand kann nur mit Funktionen geändert (`init_szeit`) oder ausgelesen (`laufdauer`) werden.

Damit sieht das Interface so aus:

laufdauer.h

[5] Im Gegensatz zur Funktion `clock` aus `time.h` wird hier die reale Zeitdifferenz ermittelt, nicht die CPU-Zeit, die das Programm verbraucht hat.

```
1 void init_szeit(void);
2 int laufdauer(void);
3 extern const int laufdauer_ver;
```

Die zwei Funktionen `init_szeit` und `laufdauer` sind also öffentlich, ebenso die Variable `laufdauer_ver`. Bei der Variablen musste das Schlüsselwort `extern` hinzugefügt werden. Es sagt dem Compiler, dass für diese Variable kein Speicherplatz reserviert werden darf. Das ist sinnvoll, denn der Speicher liegt ja in diesem Modul. Wenn also andere Module die Headerdatei `laufdauer.h` einbinden, wird nicht in jedem Modul eine solche Variable angelegt, sondern es wird immer die Variable aus `laufdauer.c` benutzt.

Und hier nun ein Beispielprogramm, mit dem man den Sinn des Moduls ausprobieren. Es entspricht dem oberen Modul in Abbildung 2.

laufdauer_bsp.c

```
1  #include <stdio.h>
2  #include "laufdauer.h"
3
4  int main(void)
5  {
6      printf("Laufdauer-Lib Version %d\n", laufdauer_ver);
7      init_szeit();
8      printf("Bitte Return-Taste druecken!");
9      getchar();
10     printf("Sie haben %d Sekunde(n) gebraucht.\n", laufdauer());
11     return 0;
12 }
```

In Zeile 4 wird die Version ausgegeben, in Zeile 7 die Initialisierung, und in Zeile 10 wird die Ermittlung der Laufdauer beauftragt und das Ergebnis ausgegeben. Das Programm kann nun gebaut und ausgeführt werden:

Terminal

```
schueler@debian964:~$ gcc -c laufdauer.c
schueler@debian964:~$ gcc -c laufdauer_bsp.c
schueler@debian964:~$ gcc laufdauer.o laufdauer_bsp.o
schueler@debian964:~$ ./a.out
Laufdauer-Lib Version 100
Bitte Return-Taste druecken!
Sie haben 3 Sekunde(n) gebraucht.
```

6.8.8 Sichtbarkeit globaler Variablen und Funktionen für den Linker

Es gibt allerdings ein kleines Problem bezüglich der Sichtbarkeit der privaten Variablen und Funktionen für den Linker. Wenn man sich das Modul `laufdauer.o` anguckt, sieht man, dass mehr Variablen drin sind als von der Headerdatei `laufdauer.h` zugegeben:

Terminal

```
schueler@debian964:~$ nm
                 U exit
0000000000000000 T fehler
                 U _GLOBAL_OFFSET_TABLE_
000000000000002b T init_szeit
0000000000000071 T laufdauer
0000000000000000 R laufdauer_ver
                 U printf
```

```
0000000000000000 D startzeit
                 U time
```

Der Befehl `nm` listet auf, was das Modul dem Linker von sich erzählt. Die drei Spalten jeder Zeile kennzeichnen die Adresse (falls bekannt), Typ und Name eines Objekts. Die Typen hier sind [6]:

U *undefined*: Eine Funktionen oder Variable, die das Modul selbst woanders herholen muss

T *text/code section*: Eine von diesem Modul bereitgestellte Funktion

R *read-only*: Eine schreibgeschützte Variable

D *initialized data section*: Eine schreibbare Variable

Mit diesen Kenntnissen könnte man natürlich `laufdauer_bsp.c` umbauen und alles, was das Modul schützen wollte, sabotieren:

laufdauer_bsp2.c

```c
#include <stdio.h>
#include "laufdauer.h"

#include <time.h>
extern time_t startzeit; // eigenmaechtig eingefuegt
void fehler(int);        // eigenmaechtig eingefuegt
int main(void)
{
    printf("Laufdauer-Lib Version %d\n", laufdauer_ver);
    init_szeit();
    startzeit=34;        // Sabotage, die erste
    printf("Bitte Return-Taste druecken!");
    getchar();
    printf("Sie haben %d Sekunde(n) gebraucht.\n",
           laufdauer());
    fehler(123);         // Sabotage, die zweite
    return 0;
}
```

Terminal

```
schueler@debian964:~$ gcc -c laufdauer.c
schueler@debian964:~$ gcc -c laufdauer_bsp2.c
schueler@debian964:~$ gcc laufdauer.o laufdauer_bsp2.o
schueler@debian964:~$ ./a.out
Laufdauer-Lib Version 100
Bitte Return-Taste druecken!
Sie haben 1575634002 Sekunde(n) gebraucht.
Zeitfehler Nr.123
```

Die Lösung liegt darin, mit einem neuen Schlüsselwort die Sichtbarkeit der privaten Variablen und Funktionen auf das Modul zu beschränken. Die Variable `startzeit` und die Funktion `fehler` müssen mit diesem Schlüsselwort versehen werden.

Hier haben die Macher der Sprache C sehr unpädagogisch gehandelt. Sie haben nämlich leider ein Schlüsselwort *wiederverwendet*, das es schon gab, nämlich das Schlüsselwort `static`. Und sie haben es mit einer neuen Bedeutung versehen. Zur Erinnerung: Innerhalb von Funktionen erhöht

[6] Weitere Typen findet man in der Manpage von `nm`.

das Attribut `static` die Lebensdauer lokaler Variablen. Hier aber geht es um globale Variablen und Funktionen[7]. Hier verringert das Attribut `static` die Sichtbarkeit dieser Variablen und Funktionen:

- Globale Variablen, die mit `static` definiert sind, sind nur im Modul sichtbar.
- Die Funktionen, die mit `static` definiert sind, sind nur im Modul sichtbar.
- Globale Variablen, die nicht mit `static` definiert sind, sind überall sichtbar (z. B. `laufdauer_ver`).
- Die Funktionen, die nicht mit `static` definiert sind, sind überall sichtbar (z. B. `init_szeit`).

Also muss man jetzt im Implementierungsteil des Moduls alle privaten Variablen und Funktionen mit `static` kennzeichnen:

laufdauer2.c

```
1 ...
2 static time_t startzeit=-1;
3
4 static void fehler(int nr)
5 ...
```

Wenn man das Modul jetzt compiliert, sieht man den Unterschied:

Terminal

```
schueler@debian964:~$ gcc -c laufdauer2.c
schueler@debian964:~$ nm laufdauer2.o
...
0000000000000000 t fehler
...
0000000000000000 d startzeit
...
```

Die Symbole `fehler` und `startzeit` werden zwar noch angezeigt, gelten jetzt aber als *privat*, erkennbar an den Kleinbuchstaben (t statt T und d statt D). Mit dieser Änderung kann die Sabotage-Version des Hauptprogramms (`laufdauer_bsp2.c`) nicht mehr ihr Unwesen treiben:

Terminal

```
schueler@debian964:~$ gcc laufdauer2.o laufdauer_bsp2.o
laufdauer_bsp2.o: In function 'main':
laufdauer_bsp2.c:(.text+0x25): undefined reference to 'startzeit'
laufdauer_bsp2.c:(.text+0x61): undefined reference to 'fehler'
collect2: error: ld returned 1 exit status
```

Als Grundregel muss man sich also nur merken: Alle globalen Variablen, Datentypen und Funktionen, die privat bleiben sollen, müssen durch das Schlüsselwort `static` geschützt werden.

Denn: `static` bei **globalen** Variablen, Datentypen und Funktionen begrenzt die Sichtbarkeit auf das aktuelle Modul.

Aber: `static` bei **lokalen** Variablen erhöht die Lebensdauer auf die Programmdauer[8].

[7] und Datentypen, aber die kommen im Beispiel nicht vor.
[8] siehe: Rückgabe von Zeigern aus Funktionen