1.1 Programmstrukturen/Unser erstes C-Programm

1.1.1 Kochrezept

Zum schnellen Einstieg gibt hier eine Anleitung:

- Start eines Editors (einer kleinen Textverarbeitung):
  Anwendungen → Zubehör → Gedit Texteditor
  Es erscheint ein leeres Fenster mit einer Menüleiste am oberen Rand.
- Den gewünschten Dateinamen festlegen:
  Menüsymbol → Speichern unter ... (oder Shift ⇑ Strg **-** S **)**
  - Name: `erstes.c` eingeben
  - Orte (=Verzeichnis): `mueller`, falls der Benutzer `mueller` heißt, also das persönliche Verzeichnis (evtl. mit einem Haus-Symbol)
  - **Speichern** anklicken
- Nun kann das Programm eingegeben werden:

src/erstes.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4     printf("Hallo, Klasse FET4U!\n");
5     return 0;
6 }
```

  Bitte achten Sie dabei exakt auf *jedes* Zeichen! Wenn man auch nur ein Semikolon vergisst oder als Doppelpunkt schreibt, läuft das Programm nicht. Ebenso ist auf Groß- und Kleinschreibung zu achten[1].
- Speichern: Der eingegebene Programmtext sollte regelmäßig gesichert werden:
  Datei → Speichern (oder Strg **-** S **)**
- Nun können Sie das Programm übersetzen in die Maschinensprache.
  - Sie öffnen ein Terminal-Fenster:
    Anwendungen → Zubehör → Terminal Das Fenster wird etwa so aussehen:

    Terminal
    ```
    schueler@debian964:~$
    ```

    Die Zeichen, die Sie bis jetzt sehen, nennt man *Eingabeaufforderung*. Sie dürfen also etwas eingeben.
  - Sie tippen: `gcc erstes.c` und drücken die Eingabetaste (die mit dem abgewinkelten Pfeil ↵):

    Terminal
    ```
    schueler@debian964:~$ gcc erstes.c
    schueler@debian964:~$
    ```

    In der obigen Darstellung ist Ihre Eingabe durch **Fettdruck** markiert. Damit kann man sie von der Eingabeaufforderung und später von den Ergebnissen Ihres Programmes unterscheiden.

[1] Etwa seit 1970 besitzen handelsübliche Terminals und Tastaturen diese Unterscheidung. Und die Programmiersprachen, die seitdem entwickelt wurden, unterscheiden ebenfalls Groß- und Kleinschreibung. Es ist also an der Zeit, sich darauf einzustellen ;-)

  - Wenn (außer der Eingabeaufforderung, die ist unvermeidlich) keine Meldung erscheint, hat alles prima geklappt.
  - Tippen Sie jetzt ein: `ls` ↩ . Wenn Sie unter anderem einen Dateinamen `a.out` sehen: Dies ist Ihr erstes fertiges Programm!

Terminal
```
schueler@debian964:~$ gcc erstes.c
schueler@debian964:~$ ls
a.out  erstes.c
schueler@debian964:~$
```

- Dieses Programm können Sie jetzt ausführen: Sie tippen – immer noch im Terminal-Fenster: `./a.out` ↩ . Es erscheint die Ausgabe Ihres Programms.

Terminal
```
schueler@debian964:~$ gcc erstes.c
schueler@debian964:~$ ls
a.out  erstes.c
schueler@debian964:~$ ./a.out
Hallo, Klasse FET4U!
schueler@debian964:~$
```

1.1.2 Editieren – Übersetzen – Ausführen: Was ist passiert?

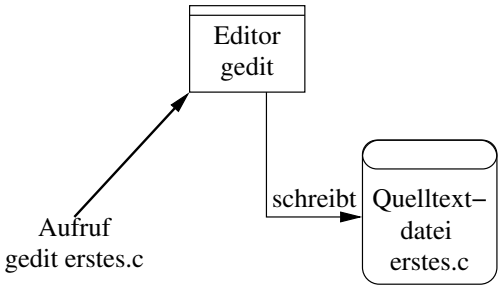

Abbildung 1: Schritt 1: Editieren

C ist eine *Programmiersprache*. In dieser Programmiersprache können wir (wie in jeder Sprache) einen Text schreiben. Bei einer Programmiersprache heißt so ein Text *Quelltext* (warum, wird gleich erklärt). Meistens legt man einen solchen Quelltext in einer Datei ab, einer Quelltext-Datei. Wir sollten Quelltext-Dateien in der Sprache C immer die Endung `.c` geben, dann finden wir sie leichter wieder.

Leider ist es so, dass die CPU eines PCs die Befehle in der Sprache C nicht versteht. Die CPU versteht nur ihre eigene Sprache, die sogenannte *Maschinensprache*. Und die CPU eines PCs spricht eine andere Maschinensprache als die CPU eines Smartphones. Und die ist wieder anders als die Maschinensprache eines Mikrocontrollers. Trotzdem – wenn Sie erst in C programmieren können, können Sie prinzipiell jedes dieser Geräte programmieren. Wie funktioniert das?

Es muss jemanden geben, der C zu lesen versteht, der aber auch mit der gerade vorhandenen CPU sprechen kann. Praktischerweise muss dies kein Mensch sein, sondern es reicht ein Dolmetscher-Programm. Dieses Programm nennt man Übersetzer oder *Compiler*. Einen solchen Compiler finden wir als ausführbare Datei (=als Programmdatei) hoffentlich auf unserer Festplatte oder SSD. Wenn wir diesen Compiler starten und ihm unsere Quelltextdatei geben, liest er sie ein. Er übersetzt die C-Anweisungen in passende Anweisungen für die CPU. Die CPU-Anweisungen schreibt er in eine andere Datei hinein, die sogenannte *Objektdatei*.

Schon sind wir fertig. Die ausführbare Datei (=Programmdatei), die der Compiler geschrieben hat, starten wir – unser Programm läuft.

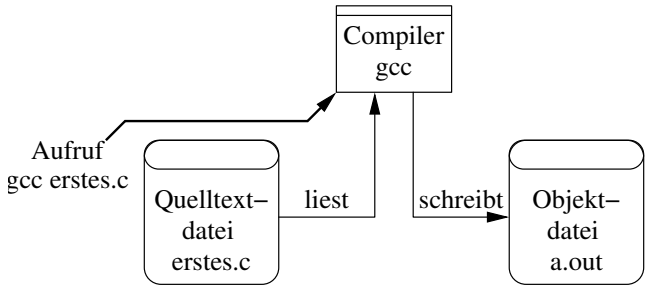

Abbildung 2: Schritt 2: Übersetzen

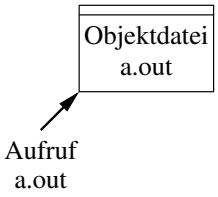

Abbildung 3: Schritt 3: Ausführen

Und jetzt wird auch klar, warum der Text, den wir in C geschrieben haben, Quelltext heißt – er ist die Quelle für den Compiler, wenn er die Programmdatei erstellt.

Das Programmieren umfasst also im Mindestfall (fehlerfreies Programm, keine Tests, keine Optimierung) drei Schritte:

a) Schreiben (=Editieren) des Quelltextes in C mit dem Editor (z. B. `gedit`) – siehe Abbildung 1

b) Übersetzen (=Compilieren) des Quelltextes hin in die CPU-Sprache mit dem Compiler (z. B. `gcc`) – siehe Abbildung 2

c) Ausführen (=*execute*) der Objektdatei (z. B. `a.out`) – siehe Abbildung 3

1.1.3 Aufbau und Funktion - Zeile für Zeile

Das Programm `erstes.c` soll nun Zeile für Zeile erklärt werden:

Zeile 1 Die Zeile `#include <stdio.h>` bindet die Datei `stdio.h` aus einem Systemverzeichnis in unser Programm ein. Sie enthält wichtige Informationen zu häufig benutzten Funktionsbausteinen.

Zeile 2 `int main(void)` ist der Einsprungspunkt des Programms: Jedes C-Programm beginnt nach diesem Wort. Es ist vergleichbar mit dem Organisationsbaustein `OB1` bei manchen SPS.

Mit `int main(void)` beginnt ein sogenannter Funktionsbaustein (abgekürzt: Funktion). Wenn man einen Funktionsbaustein benutzt, liefert er einen Wert zurück – es gibt allerdings auch Funktionsbausteine, die das nicht tun. Z.B. liefert der Funktionsbaustein `time` einen Wert den Wert zurück, der über die aktuelle Uhrzeit Auskunft gibt.

Der Funktionsbaustein `main` jedenfalls liefert an den, der ihn benutzt, eine Zahl zurück. Dafür steht das Wort `int`.

Die runden Klammern einer Funktion können sogenannte Parameter enthalten, das ist etwa beim Aufruf `sin(x)` der Wert `x`. Beim Funktionsbaustein `main` ist (in diesem Programm) so ein Parameter vorgesehen. Die Klammer ist leer. Dafür steht das Wort `void`.

Zeile 3 Zwischen den geschweiften Klammern steht der gesamte Inhalt der Funktion. Er besteht in der Regel aus Anweisungen, häufig auch aus Vereinbarungen.

Wenn ich das Programm ausführe (=laufen lasse), werden die Anweisungen ausgeführt, die zwischen den geschweiften Klammern des Funktionsbausteins `main` stehen.

Zeile 4 `printf("text")` ist die erste Anweisung. Wenn ich das Programm ausführe, wird sie als erste ausgeführt (viel mehr Anweisungen hat unser Programm ja noch nicht).

In dieser Zeile wird der Funktionsbaustein `printf` benutzt, man sagt, *die Funktion printf wird aufgerufen*. Der Funktionsbaustein `printf` dient der Bildschirmausgabe von Texten (und Zahlen, dazu später).

Wenn diese Anweisung ausgeführt wird, gibt der Computer auf den Bildschirm den Text aus, der zwischen den beiden Anführungszeichen steht. Dieser Text in Anführungszeichen heißt *Zeichenkette* bzw. *String*.

Am Ende der Zeichenkette steht die merkwürdige Zeichenfolge `\n`. Sie taucht aber gar nicht auf dem Bildschirm auf. Bei `\n` handelt es sich nämlich um eine sogenannte *Ersatzdarstellung* eines Sonderzeichens. `\n` bedeutet den Zeilenvorschub (*new line*), an dieser Stelle springt der Cursor (=Textzeiger) in die nächste Zeile. Eine solche Ersatzdarstellung darf nur in Zeichenketten stehen.

Zum Schluss der Anweisung findet man ein Semikolon. Das Semikolon beendet eine Anweisung. Gleichzeitig ist ein Semikolon ein Wartepunkt: Erst wenn eine Anweisung fertig ist, kann mit der nächsten Anweisung begonnen werden.

Zeile 5 `return 0;` ist eine Anweisung zum Beenden des Funktionsbausteins. Wo sie steht, wird der Funktionsbaustein beendet, im Fall von `main` das ganze Programm. Außerdem gibt sie an das aufrufende Programm (z.B. Konsole, cmd, graphische Oberfläche) die Zahl zurück, die hinter `return` steht. Die Zahl 0 bedeutet, dass das Programm Erfolg meldet. Jede andere Zahl meldet einen Fehler zurück, wobei man sich die Zahlen als Programmierer frei aussuchen kann[2].

Zeile 6 Mit der schließenden geschweiften Klammer ist der Funktionsbaustein `main` abgeschlossen.

1.1.4 Erweitern des Programmes

Vor der `return 0;`-Zeile kann man weitere `printf()`-Anweisungen einfügen. Jede Anweisung muss durch ein Semikolon abgeschlossen werden! Man erhält dadurch eine *Sequenz* (=Abfolge) von Anweisungen. Sequenz bedeutet, dass die Aweisungen nacheinander ausgeführt werden. Erst wenn die erste Anweisung fertig ist, wird die zweite begonnen.

Die Sequenz ist eine von mehreren grundlegenden *Programmstrukturen*. Sie gehört damit zur Grammatik der Programmiersprache C.

src/zweites.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4     printf("Dies ist schon mein zweites Programm.\n");
5     printf("Es enthaelt mehrere Zeilen.\n");
6     printf("Unglaublich.\n");
7     return 0;
8 }
```

Nun kann man das Programm compilieren und ausführen:

[2]Seit C11 gibt es für den Funktionsbaustein `main` die Sonderregel, dass die Anweisung `return 0;` am Schluss des Funktionsbausteins weggelassen werden darf.

Terminal

```
schueler@debian964:~$ gcc zweites.c
schueler@debian964:~$ a.out
Dies ist schon mein zweites Programm.
Es enthaelt mehrere Zeilen.
Unglaublich.
```

Oft besteht der Wunsch, den Aufbau eines Programms graphisch darzustellen. Die bekannteste Möglichkeit dazu ist das Flussdiagramm, auch Programm-Ablauf-Plan (PAP) genannt (Abb. 4, Mitte). Hier beginnt der Programmablauf mit dem Symbol *Start* und endet mit dem Symbol *Stop*. Zwischen Start und Stop folgt der Programmablauf den Pfeilen. Pfeilspitzen von oben nach unten und von links nach rechts dürfen weggelassen werden.

Eine andere Darstellung ist das Struktogramm, nach seinen Erfindern auch Nassi-Shneidermann-Diagramm genannt (Abb. 4 links). Hier beginnt der Programmablauf am oberen Rand eines Rechtecks und endet am unteren Rand.

Schließlich gibt es noch das Jackson-Diagramm (Abb. 4 rechts). Hier werden die nacheinander ablaufenden Programmteile als Baumdiagramm nebeneinander dargestellt.

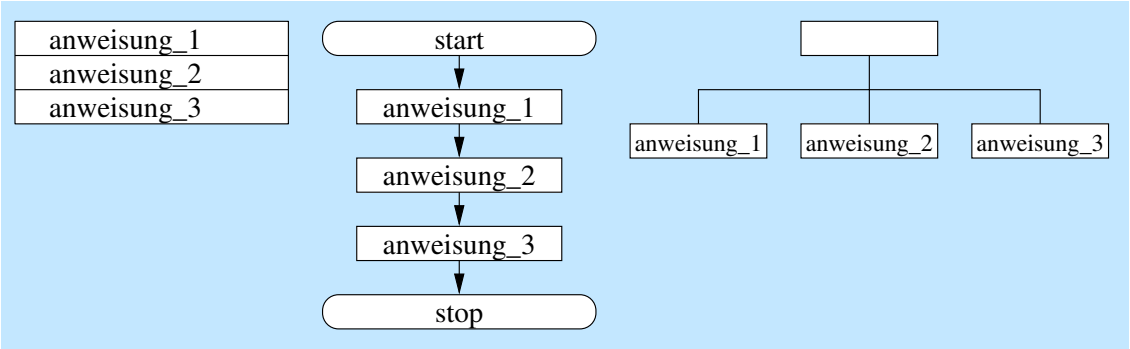

Abbildung 4: Sequenz in drei verschiedenen Diagrammformen

Abbildung 5 zeigt, wie das Struktogramm zu `zweites.c` aussehen kann, und Abbildung 6 zeigt das entsprechende Flussdiagramm. Im Flussdiagramm sind Anweisungen allgemein als Rechtecke gezeichnet; Ein- und Ausgabeanweisungen darf man als Parallelogramme zeichnen.

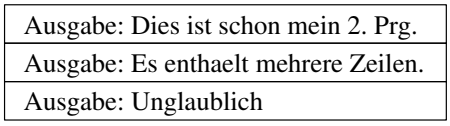

Abbildung 5: Struktogramm von `zweites.c`

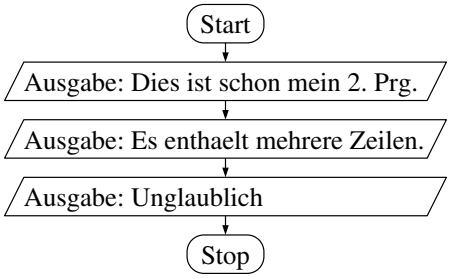

Abbildung 6: Flussdiagramm von `zweites.c`

1.1.5 Kommentare

Manchmal möchte man im Quelltext Anmerkungen oder Notizen hinzufügen. Dazu gibt es in vielen Programmiersprachen *Kommentare*. Ein Kommentar ist ein Abschnitte im Quelltext, den der Compiler überliest oder gar nicht erst zu sehen bekommt. Das gibt es auch in C.

src/drittes.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4      /*
5      Zum Anfang eine erste
6      printf-Anweisung    */
7
8      printf("Hallo!\n");
9      // Danach eine zweite-printf-Anweisung
10     printf("Hier waeren wir\n");
11     return 0;
12 }
```

Zeile 4 Hier beginnt ein Kommentar.

Zeile 6 Und hier endet er.

Zeile 7 Leerzeilen sind erlaubt. Auch sie dienen der Übersichtlichkeit.

Zeile 9 Dieser Kommentar geht nur bis zum Ende der Zeile (ab C99).

1.1.6 Längere Zeichenketten

In C sind auch längere Zeichenketten kein Problem. Allerdings darf eine einzelne Zeichenkette *im Quelltext* nicht über mehrere Zeilen gehen. Das folgende Programm wird deshalb nicht compiliert:

src/viertes.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Ein ziemlich langer Text, deshalb mache ich
5  ␣␣␣␣␣␣␣␣␣␣es so\n");
6      return 0;
7  }
```

Beim Compilieren passiert dies:

Terminal

```
schueler@debian964:~$ gcc viertes.c
viertes.c: In function 'main':
viertes.c:4:11: warning: missing terminating " character
    printf("Ein ziemlich langer Text, deshalb mache ich
           ^
... weitere Warnungen und Fehlermeldungen ...
```

Man kann aber eine Zeichenkette aufteilen in mehrere aufeinanderfolgende; der Compiler[3] fügt sie wieder zusammen:

src/fuenftes.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Ein ziemlich langer Text, deshalb mache ich "
```

[3] genauer gesagt, seine Eingangsstufe, der Präprozessor

```
5           "es so\n");
6   return 0;
7   }
```

Das gibt die Ausgabe:

Terminal

```
schueler@debian964:~$ gcc fuenftes.c
schueler@debian964:~$ a.out
Ein ziemlich langer Text, deshalb mache ich es so
```

Wenn man den Text *bei der Ausgabe* in mehrere Zeilen aufteilen will, dann kann man das durch zusätzliche Zeilenvorschübe hinbekommen:

src/sechstes.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4      printf("Ein ziemlich langer Text,\ndeshalb mache ich\n"
5             "es so\n");
6      return 0;
7  }
```

Das gibt die Ausgabe:

Terminal

```
schueler@debian964:~$ gcc fuenftes.c
schueler@debian964:~$ a.out
Ein ziemlich langer Text,
deshalb mache ich
es so
```

1.1.7 Ergänzungen

Hier sollen noch zwei weitere einfache Funktionsbausteine vorgestellt werden. Die Funktion `sleep` sorgt dafür, dass der Programmlauf an dieser Stelle eine oder mehrere Sekunden wartet und dann erst weitergeht:

src/siebtes.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main(void)
4  {
5      printf("gruen\n");
6      sleep(5);
7      printf("gelb\n");
8      sleep(2);
9      printf("rot\n");
10     return 0;
11 }
```

Zeile 2 Für den Funktionsbaustein `sleep` muss die Datei `<unistd.h>` eingebunden werden. Das geschieht mit dieser Zeile. Leider ist `sleep` nicht im C-Standard enthalten. Bei anderen Systemen kann `sleep` fehlen oder eine andere Funktionalität haben (z. B. ist der Parameter dann die Wartezeit in Millisekunden).

Zeile 6 `sleep` erwartet in der Parameterliste (=in den runden Klammern) eine Zahl. Diese Zahl gibt an, wie viele Sekunden an dieser Stelle gewartet wird. In diesem Fall sind es 5 Sekunden, die hier gewartet wird.

Zeile 8 Hier wird 8 Sekunden gewartet.

Zeile 10 Und hier wird 10 Sekunden gewartet.

Die Funktion `system` sorgt dafür, dass im Programm ein anderes Programm aufgerufen wird:

src/achtes.c

```
1  #include <stdio.h>  // fuer printf
2  #include <stdlib.h> // fuer system
3  #include <unistd.h> // fuer sleep
4  int main(void)
5  {
6      printf("Erst kommt die Uhrzeit:\n");
7      sleep(2);
8      system("date");
9      sleep(2);
10     printf("Jetzt kommt der Kalender:\n");
11     sleep(2);
12     system("cal");
13     return 0;
14 }
```

Zeile 2 Für den Funktionsbaustein `system` muss die Datei `<stdlib.h>` eingebunden werden.

Zeile 7 Die `sleep`-Aufrufe in diesem Programm sind reine Effekthascherei.

Zeile 8 Der Konsolen-Befehl `date` wird aufgerufen.

Zeile 12 Der Konsolen-Befehl `cal` wird aufgerufen.

Man kann mit `system` auch graphische Programme aufrufen:

src/neuntes.c

```
1  #include <stdio.h>  // fuer printf
2  #include <stdlib.h> // fuer system
3  #include <unistd.h> // fuer sleep
4  int main(void)
5  {
6      printf("Erst kommt der Browser:\n");
7      sleep(2);
8      system("firefox");
9      sleep(2);
10     printf("Jetzt kommt der Taschenrechner:\n");
11     sleep(2);
12     system("gnome-calculator");
13     return 0;
14 }
```

Zeile 8 Der Webbrowser `firefox` wird aufgerufen.

Zeile 12 Der Taschenrechner `gnome-calculator` wird aufgerufen.

Sollen die graphischen Programme im Hintergrund weiterlaufen, so wird an den Befehl das Zeichen & angehängt:

src/zehntes.c

```
1  #include <stdio.h>  // fuer printf
2  #include <stdlib.h> // fuer system
3  #include <unistd.h> // fuer sleep
4  int main(void)
5  {
6      printf("Augen Nr. 1:\n");
7      system("xeyes &");
8      sleep(2);
9      printf("Augen Nr. 2:\n");
10     system("xeyes &");
11     sleep(2);
12     printf("Augen Nr. 3:\n");
13     system("xeyes &");
14     return 0;
15 }
```

Zeile 7 Das Programm `xeyes` wird im Hintergrund aufgerufen.

Zeilen 9 und 11 Dasselbe Program wird noch zweimal im Hintergrund aufgerufen.