Programmieren I

Programmieren in C

WS 2024

1 Einführung

C ist eine imperative Programmiersprache, die im Jahr 1972 erstmals veröffentlicht wurde. Da C sehr maschinennah arbeitet und somit wenig Abstraktion beinhaltet, wird sie oft für Betriebssysteme, Treiber oder embedded Systeme wie Arduinos, RasPis oder Autos verwendet. Man betitelt C daher auch als **Low-Level Sprache**.

Die fehlende Abstraktion und der maschinennahe Ansatz sorgt in C dafür, dass ein Entwickler wenige Beschränkungen hat. Prinzipiell macht C-Code genau das, was der Entwickler ihm sagt. Dies gibt eine immense Freiheit, aber bringt auch ein gewisses Risiko mit sich, da durch die lockeren Compilerbeschränkungen auch viele Fehler passieren können.

Beispiel 1.1. Ein Entwickler legt ein Array mit einer festen Länge (z.B. 10) an. Um auf ein Element dieses Arrays zuzugreifen, wird die Syntax `array[i]` verwendet, wobei `i` der Index (also die Stelle) des Elements, das abgerufen werden soll, ist. Der Index beginnt bei 0 und endet bei der Länge des Arrays minus 1. Für dieses Beispiel würde das bedeuten, dass der höchste, im Array enthaltene Index 9 ist. Ein unerfahrener Entwickler könnte nun allerdings versuchen, das Array an der Stelle `array[10]` aufzurufen, da er denkt, dass der Index bei 1 beginnt und das letzte Element den Index 10 besitzt. Was nun?

In vielen anderen Programmiersprachen würde das Programm nicht kompilieren, da der Compiler bereits erkennt, dass der Index zu groß ist und außerhalb des Arrays liegt. Wie allerdings bereits in der Einführung erwähnt, gibt C dem Entwickler die volle Kontrolle über sein Handeln und kompiliert den Code trotzdem. Das führt dazu, dass wenn das Programm zur Laufzeit auf das Element `array[10]` zugreifen möchte, es auf den Speicherbereich zugreift, der nach dem Array liegt. Dort können Daten liegen, dort können aber auch einfach leere Speicherzellen sein. In jedem Fall greift das Programm jedoch auf einen Speicherbereich zu, auf den es nicht zugreifen sollte. Je nachdem, welcher Bereich das ist, kann es ebenfalls passieren, dass das Betriebssystem das Programm beendet.

Dieses Beispiel soll verdeutlichen, dass C eine mächtige Programmiersprache ist, bei der dem Entwickler viel Freiheit geboten wird, aber auch viel Verantwortung auf ihm lastet. Es ist daher wichtig, sorgfältig und bedacht in C zu programmieren und bewusst mit den Freiheiten umzugehen.

2 Hello, World!

Beginnt man mit einer völligen neuen Programmiersprache, ist der erste Code, das man in dieser Sprache entwickelt, oft ein simpler Textoutput in der Konsole.

Möchte man in C sieht ein solches Programm schreiben, könnte das wie folgt aussehen:

```c
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

Fig. 1: Hello, World! in C

Anhand dieses Beispiels lassen sich schon einige Aspekte der generellen C-Syntax erkennen. Zum Beispiel ist die Verwendung von Semikolon (`;`) am Ende einer Anweisung erkennbar sowie auch die Verwendung von geschweiften Klammern ({ }) für Codeblöcke (Scopes). Die genauen Bedeutungen der einzelnen Bestandteile werden nach und nach in den nächsten Abschnitten erläutert.

3 Datentypen, Variablen und Operatoren

Ein elementarer Bestandteil von Programmiersprachen ist die Fähigkeit, Daten temporär zu speichern. Damit ein Programm dem Betriebssystem die Aufgabe geben kann, Daten temporär zu speichern, benötigt es jedoch zwei Informationen:

- **Welche** Art von Daten soll gespeichert werden?
- **Wie groß** sind die Daten, bzw. **wie viel** Speicherplatz nehmen sie ein?

In C werden diese Informationen durch **Datentypen** und **Variablen** definiert.

3.1 Datentypen

Bevor wir zu Datentypen kommen, muss erstmal geklärt werden, wie Daten überhaupt dargestellt werden.

Ein Computer speichert Daten in Form von **Bits** (Binary Digits), also einer Abfolge von Nullen und Einsen. Das heißt, die Zahl 25 kann einfach in ihrer binären Form als 11001 im Speicher abgelegt werden.

Nun ist allerdings das Problem, dass wir nicht wissen, für was ein Datensatz 11001 im Speicher steht. Nur weil wir wissen, dass dieser Datensatz die Dezimalzahl 25 in Binärform darstellt, heißt das aber nicht, dass auch genau das damit gemeint ist. Es könnte sich auch um ein oder mehrere Zeichen halten, die mittels einer bestimmten Codierung (z.B. ASCII) gespeichert wurden.

Um dieses Problem zu lösen, gibt es in C sogenannte **Datentypen**. Ein Datentyp definiert, wie eine Abfolge von Bits von einem Programm interpretiert werden soll. Dabei gibt ein Datentyp neben der "Art" der Daten auch an, wie viel Speicherplatz diese Daten einnehmen. Es gibt eine Vielzahl an verschiedenen Datentypen. Einige davon zeigt Tabelle 1.

Datentyp	Keyword	Speichergröße	Wert
Integer	int	16 bit / 32 bit	Ganze Zahlen
Short	short	16 bit	Ganze Zahlen
Long	long	32 bit	Ganze Zahlen
Long Long	long long	64 bit	Ganze Zahlen
Float	float	32 bit	Gleitkommazahlen
Double	double	64 bit	Gleitkommazahlen
Character	char	8 bit	Ganze Zahlen

Table 1: Primitive Datentypen in C

Anmerkung 3.1. Alle ganzzahligen Datentypen sind vorzeichenbehaftet, d.h. sie können sowohl positive als auch negative Werte speichern. Das bedeutet aber auch, dass ein Bit vom Datenbereich für das Vorzeichen reserviert ist und sich der Wertebereich somit um eine Potenz von 2 verringert (z.B. bei einem 16-bit Integer von -2^{15} bis $2^{15} - 1$).

Möchte man dieses Bit lieber nutzen, um den Wertebereich zu erhöhen und dafür auf das Vorzeichen verzichten, gibt es die Möglichkeit den Datentypen mit dem Keyword `unsigned` zu versehen. Ein `unsigned int` kann also Werte von 0 bis $2^{16} - 1$ speichern.

3.2 Variablen

Wir haben nun Datentypen kennengelernt, aber wie verwenden wir sie in einem Programm? Dafür benötigen wir **Variablen**.

Eine Variable ist im Endeffekt nichts anderes als ein reservierter Speicherbereich, in dem ein Programm während der Laufzeit Daten ablegen und auslesen kann. In C ist einer Variable immer ein fester Datentyp zugewiesen, welcher sich auch nach der Deklaration nicht mehr ändern lässt. Die Syntax zur Erstellung einer Variable lautet:

```
<Datentyp> <Variablenname>;
```

Anmerkung 3.2. Als **Deklaration** bezeichnen wir die Bekanntmachung einer Variable an den Compiler. Das bedeutet, der Compiler weiß nun, dass wir eine Variable mit einem bestimmten Namen erstellen wollen und reserviert dafür entsprechend des Datentyps den Speicherplatz. Bei der Deklaration muss der Variable aber noch nicht zwingend ein Wert zugewiesen werden. Das erfolgt bei der Initialisierung, also der Zuweisung eines Wertes zu einer Variable.

Beispiel 3.1. Um eine Variable `a` vom Typ `int` zu deklarieren, verwenden wir in C den Aufruf:

```
int a;
```

Wie bereits erwähhnt, wird der Variable durch eine reine Deklaration noch kein Wert zugewiesen. Würden wir die Variable nun so aufrufen, würde sie den Wert zurückgeben, der noch im Speicher an der Stelle vorhanden ist. Daher sollte eine Variable immer bei der Deklaration **initialisiert** werden, um unerwünschtes Verhalten zu vermeiden:

```
int a = 0;
```

3.3 Operatoren

Wir haben nun also die Möglichkeit Daten zu speichern und festzustellen, um welche Art von Daten es sich handelt. Nun wäre es auch sinnvoll, mit diesen Daten zu arbeiten. Dafür gibt es in C die sogenannten **Operatoren**.

Ein **Operator** verknüpft Variablen miteinander und führt eine Operation auf ihnen aus. Es gibt dabei verschiedene Arten von Operatoren, die unterschiedliche Aufgaben erfüllen. Eine davon kennen wir bereits aus der Mathematik, nämlich die **arithmetischen Operatoren**. Diese dienen, wie bereits der Name sagt, dazu, Rechenoperationen auf Variablen auszuführen. In C gibt es folgende arithmetische Operatoren:

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo (Rest einer Division)

Table 2: Arithmetische Operatoren in C

Anmerkung 3.3. Arithmetische Operatoren sollten nur auf Variablen angewendet werden, die als Zahl interpretiert werden sollen. Ein Character (`char`) wird intern auch als Zahl gehandhabt und somit kann man mit ihnen auch rechnen, nur wird dabei eventuell nicht das gewünschte Ergebnis erziehlt.

Beispiel 3.2. Gegeben sind zwei ganzzahlige Variablen `a` und `b` und wir möchten den Rest der ganzzahligen Division von `a` durch `b` berechnen. Der Code dazu könnte wie folgt aussehen:

```
int a = 10;
int b = 3;
int rest = a % b; // = 1, da 10/3 = 3 Rest 1
```

An diesem Beispiel erkennen wir, dass das Ergebnis einer arithmetischen Operation eine Zahl ist, die wir beispielsweise in einer neuen Variable speichern können.

Neben den arithmetischen Operatoren existieren darüber hinaus auch die logischen Operatoren. Diese Operatoren geben einen Wahrheitswert (0 für falsch, 1 für wahr) zurück und werden häufig in Verbindung mit **Bedingungen** (s. Kapitel 4) verwendet. In C gibt es folgende logische Operatoren:

Operator	Bedeutung
&&	logisches Und
||	logisches Oder
!	logisches Nicht

Table 3: Logische Operatoren in C

Die dritte Art von Operatoren sind **Vergleichsoperatoren**. Diese verwendet man, um zwei Werte direkt miteinander zu vergleichen und geben daher ebenfalls einen Wahrheitswert zurück. In C gibt es folgende Vergleichsoperatoren:

Operator	Bedeutung
==	Gleich
! =	Ungleich
>	Größer
<	Kleiner
>=	Größer oder gleich
<=	Kleiner oder gleich

Table 4: Vergleichsoperatoren in C

Wichtig: `=` und `==` sind zwei verschiedene Operatoren! Das einfache Gleichheitszeichen wird verwendet, um einer Variable einen Wert zuzuweisen. Möchte man den Wert einer Variable mit einem anderen Wert auf Gleichheit prüfen, verwendet man dazu immer ein doppeltes Gleichheitszeichen.

Ein entsprechendes Codebeispiel zur Anwendung von Vergleichsoperatoren folgt auch hier in Kapitel 4.

4 Kontrollstrukturen

Ein Programm, das lediglich aus Variablen und Operatoren besteht, wäre relativ sinnfrei, da es keine Möglichkeit gäbe, den Verlauf des Programms auf die Eingaben des Benutzers oder auf andere Gegebenheiten anzupassen. Das könnte zum Beispiel die Berechtigung eines Benutzers sein; ein Programm soll vielleicht andere Funktionen ausführen, wenn es sich bei dem Benutzer um einen Administrator handelt.

Um solche Bedingungen in den Verlauf des Programms einzubeziehen, gibt es sogenannte **Kontrollstrukturen**. Grundsätzlich lassen sich diese in zwei Kategorien unterteilen: **Bedingte Anweisungen** und **Schleifen**.

4.1 Bedingte Anweisungen

Eine bedingte Anweisung ist ein Codeblock, der nur dann ausgeführt wird, wenn eine bestimmte Bedingung erfüllt wird. Das Keyword in C dafür lautet `if` und ist wie folgt definiert:

```
if (<Bedingung>) {
  // Code, der ausgeführt wird, wenn die Bedingung erfüllt ist
}
```

Fig. 2: if-Statement in C

Beispiel 4.1. Gegeben ist eine Variable `alter`, die das Alter des Benutzers speichert. Das Programm soll nun einen Text ausgeben, wenn der Benutzer volljährig ist. Der Code dazu könnte wie folgt aussehen:

```
int main() {
  int alter = 18;
  if (alter >= 18) {
    printf("Du bist volljährig!\n");
  }

  return 0;
}
```

Führt man dieses Programm nun aus, erscheint der Text `Du bist volljährig!` in der Konsole. Ändert man den Wert der Variable `alter` aber beispielsweise auf `17`, passiert gar nichts. Das liegt daran, dass der Code innerhalb des `if`-Statements wirklich nur dann ausgeführt wird, wenn die Bedingung in den runden Klammern erfüllt ist.

Um das Programm dahingehend zu erweitern, dass auch ein Text erscheint, wenn der Benutzer nicht volljährig ist, gibt es in C das Keyword `else`. Der Code innerhalb eines else-Blocks wird immer dann ausgeführt, wenn die Bedingung im vorangestelltn `if`-Statement nicht erfüllt ist. Für das obige Beispiel könnte ein passender `else`-Block wie folgt aussehen:

```
int main() {
  int alter = 17;
  if (alter >= 18) {
    printf("Du bist volljährig!\n");
  } else {
    printf("Du bist minderjährig!\n");
  }

  return 0;
}
```

5 Funktionen