Die Sprachstandards

Lars Thoms

Inhaltsverzeichnis

1 Einleitung

Ein Sprachstandard, wie z.B. der C-Standard C99, ist eine Vereinbarung, welche getroffen wurde, um eine einheitliche Sprache zu gestalten.

Bevor die Sprachstandards in der Sprache C ratifiziert wurden, gab es sehr viele Gruppen, welche C programmiert bzw. entwickelt haben. Dadurch entstanden unter anderem unterschiedliche Syntaxen, Standardbibliotheken und Arbeitsabläufe.

Um einen einheitlichen Compiler für alle C-Programme zu konzipieren, ist es notwendig, dass die akzeptierte Sprache des Compilers standardisiert wird. Somit ist es möglich, alle C-Programme mit einem Compiler zu erzeugen.

Ein klassicher Aufbau eines C-Programms:

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

int main(int argc, char *argv[])
{
  printf("Hello World!");
  exit(EXIT_SUCCESS);
}
```

1.1 Die Sprachstandards

Im Laufe der Zeit entstanden und entstehen neue, überarbeitete Versionen des C-Sprachstandards. Die ersten Standards wurden von der amerikanischen Normierungsbehörde »ANSI[1]« in einer Arbeitsgruppe erarbeitet und ratifiziert.

Die Standards der ANSI, die heute noch teilweise im Einsatz sind:

- C89
- C90

Die Ziffern nach dem »C« geben die Jahreszahl der Ratifizierung des Standards an.

[1] American National Standards Institute

Alle folgenden C-Standards werden im Zuge der Globalisierung von einer Arbeitsgruppe der »ISO[2]« beschlossen. Die ISO hat bisher drei weitere Standards veröffentlicht, wobei der letzte noch nicht großflächig genutzt wird:

- C95
- C99
- C11

1.2 Destruktivität

Wenn ein neuer Sprachstandard beschlossen wird, ist das Hauptziel immer den aktuellen Standard zu erweitern. Die akzeptierte Sprache darf dabei auch nur erweitert werden, damit kein Programm neu geschrieben werden muss, weil sich der Standard ändert.

Dadurch wird auch sichergestellt, dass Lösungen für bestimmte Probleme – mögen sie noch so abwegig erscheinen – nach einer Aktualisierung immer noch funktionieren.

2 Die Syntax

2.1 Grundlegendes

Der Sprachstandard sieht für die Sprache C vor, dass nach jedem Befehl ein Semikolon stehen muss. Ein Befehl kann beispielsweise ein Funktionsaufruf oder eine Variablendeklaration sein.

```
foo(arg1, arg2, ...); //Funktionsaufruf

int zahl = 42;          //Variablendeklaration
```

Des Weiteren werden Codeblöcke mit geschweiften Klammern eingerahmt:

```
if( zahl == 42 )
{
  ...
}
```

[2] International Organization for Standardization

2.2 Variablen

Variablen bilden eine wichtige Komponente einer imperativen Programmiersprache. Sie verweisen auf formatierte Daten im Stack oder Heap und der Wert kann zur Laufzeit verändert werden.

Eine Variable hat immer einen *Typ*, einen *Namen* und einen *Wert*:

```
type name = value;

int  zahl = 42;
```

Wo Variablen deklariert werden müssen, hängt ganz davon hab, wie ihre Lebenszeit aussehen soll und welchen Standard man benutzt.

Eine Variable, die auf dem Stack liegt, ist solange gültig, wie ihr Codeblock, in dem sie deklariert wurde. Dagegen behält eine Variable auf dem Heap so lange ihre Gültigkeit, bis man den allokierten Speicher wieder freigibt oder das Programm terminiert.

Position der Deklaration Der Standard C89 sieht vor, dass alle Variablen in einem zusammenhängenden Codeblock am Beginn zu deklarieren sind.

```
{
  int zahl = 42;
  int i, output;

  for(i=0; i < zahl; i++)
  {
    ...
  }

  output = 13;

  ...
}
```

Ab dem Standard C99 ist es nicht mehr notwendig, diese strikte Regelung einzuhalten.

```
{
  int zahl = 42;
  int i;

  for(i=0; i < zahl; i++)
  {
    ...
  }

  int output = 13;

  ...
}
```

2.3 Konstanten

Konstanten sind keine Sonderform von Variablen. Sie verweisen zu keiner Zeit auf den Speicher (Stack oder Heap). Der Compiler ersetzt die Konstante im Programmcode durch den Wert, der ihr zugewiesen wurde.

Dementsprechend ist eine Konstante auch zur Laufzeit nicht mehr veränderbar, da der Wert fest im Programmcode verankert ist.

```
const type name = value;

const int  foo  = 42;
```

3 Datentypen

Datentypen definieren in einer Programmiersprache die Art der Speicherrepräsentation und die Interpretation der Daten.

Ein *char* beispielsweise ist nichts anderes als eine Zahl. Man könnte also mit *chars* ganz normale mathematische Operationen durchführen. Erst durch die Festlegung es als Zeichen zu interpretieren, erfüllt es seinen Zweck.

Die Datentypen beanspruchen auch unterschiedliche Speichergrößen, die sich nach Architektur/System unterscheiden können.

3.1 Ganze Zahlen

Eine ganze Zahl wird im C-Standard als *integer* bezeichnet.

Der Standard schreibt vor, dass *int* mindestens 2 Byte und char genau 1 Byte breit sein müssen.

Dabei muss folgende Bedingung erfüllt sein:
signed char $\leq$ *short int* $\leq$ *int* $\leq$ *long int* $\leq$ *long long int*

Folgend sieht man eine Tabelle[3] mit den bekanntesten Datenmodellen, ihren jeweiligen *int*-Größen und ihre Einsatzgebiete:

Modell	char	short	int	long	long long	Plattform
IP16	8	16	16	32	64	MS DOS im SMALL memory model
LP32	8	16	16	32	64	MS DOS im LARGE memory model
ILP32	8	16	32	32	64	x86-System
LLP64	8	16	32	32	64	Windows x86-64
LP64	8	16	32	64	64	Unix/Linux x86-64
ILP64	8	16	64	64	64	SPARC64

3.2 Gleitkommazahlen

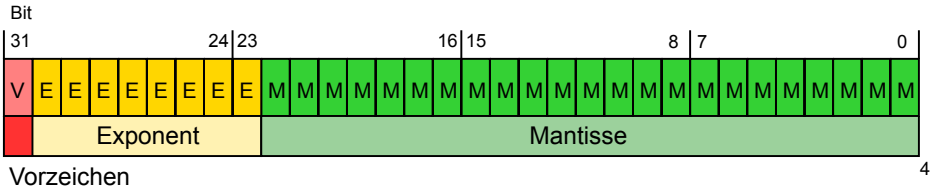
Gleitkommazahlen werden via *float* deklariert. Dabei gibt es üblicherweise noch *double* und *long double* mit einem größeren Speicherbereich.

Wenn man den Speicherbereich für den Exponenten erhöht, dann ist es möglich, noch größere oder kleinere Zahlen darzustellen. Wenn man hingegen den Speicherbereich der Mantisse erhöht, dann ist die Genauigkeit der Zahl höher, da mehr Nachkommastellen abgebildet werden können.

```
float foo = 1.0;
float bar = 1f;
```

Der Standard schreibt dagegen nicht vor, welches Speicherabbild für die Gleitkommazahlen verwendet werden soll, empfielt aber das *IEEE 754-Format*.

[3] Quelle: http://de.wikipedia.org/wiki/C_%28Programmiersprache%29#Datenmodell



3.3 Sonstige Primitive Datentypen

Komplexe Zahlen Ab dem Standard C99 wurden auch komplexe Zahlen in der Sprache C spezifiziert. Dabei erweitert man den Datentyp *float* um das Schlüsselwort *complex*.

- *float complex*
- *double complex*
- *long double complex*

Wichtig ist hierbei, dass der Header *complex.h* benötigt wird.

Logische Typen In C sind die boolesche Werte *true* und *false* nicht standardmäßig vorhanden. Erst durch den Header *stdbool.h* kann man diesen Datentyp nutzen.

3.4 Arrays

Arrays sind nichts anderes, als eine mehrfache Aneinanderreihung eines Datentyps. Dabei kann man über einen Index direkt darauf zugreifen, da die Größe des Datentyps bekannt ist und somit der Offset bis zum entsprechenden Speicherbereich berechenbar ist.

Char-Array / Zeichenkette:

```
char foo[12] = "Hello World!";

printf("%s", foo[6]); //Output: W
```

Dabei ist man nicht beschränkt auf eindimensionale Arrays –
C kann n-dimensionale Arrays abbilden.

Ein Beispiel für ein zweidimensionales *int*-Array:

[4]Quelle: http://upload.wikimedia.org/wikipedia/commons/5/56/IEEE-754-single.svg

```
int foo[2][3] = {{13,3,7},{0,4,2}};

foo[0][0] = 1;
foo[0][1] = 33;

printf("%d", foo[1][2]); //Output: 2
```

3.5 Aufzählung »enum«

C bietet die Möglichkeit eine Reihe von Konstanten automatisch zu nummerieren. Dadurch spart man sich die Zeit jede Konstante einzeln zu deklarieren.

In dem Beispiel nummerieren wir eine komplette Woche und setzen den Startwert auf »1«:

```
enum week
{
  Mon=1, Tue, Wed, Thu, Fri, Sat, Sun
};

printf("%d", Fri); //Output: 5
```

3.6 Struktur »struct«

Der C-Standard stellt mehrere Datentypen bereit. Jedoch ist es manchmal nötig eigene Datentypen zu erstellen. Vor allem, wenn man ein komplexes Objekt abbilden möchte.

Dafür gibt es in C das *struct*. Mit ihm kann man beispielsweise Variablen in einem Block zusammenfassen und es als neuen Datentypen benutzen.

In diesem Beispiel wird ein neuer Datentyp *Foo* erstellt. Er beinhaltet drei Variablen vom Typ *int*. Zugriff auf die »untergeordneten« Variablen bekommt man unter anderem durch eine Punktnotation.

```
struct Foo
{
  int a, b, c;
};

struct Foo bar;
bar.a = 42;
```

3.7 Typumwandlung

Hin und wieder ist es erforderlich, bestimmte Werte in einen anderen Typ umzuwandeln oder eine Typzusicherung durchzuführen. Dafür schreibt man in C vor dem Ausdruck, den man umwandeln möchte, in runden Klammern den neuen Datentyp.

Jedoch verliert man dadurch fast immer Informationen. Bei der Umwandlung von einer Gleitkommazahl in eine ganze Zahl wird einfach der Nachkommabereich abgeschnitten. Bei der Umwandlung von einer ganzen Zahl in eine Gleitkommazahl, die beispielweise im IEEE-754-Format abgebildet wird, kann es zu einer Ungenauigkeit kommen.

```
float a = 10.75;
int b = (int) a;
printf("%d", b); //Output: 10
```

4 Operatoren

4.1 Arithmetische Operatoren

Durch arithmetische Operatoren kann man den Wert von Variablen verändern. Dabei ist die Operatorenpräzedenz zu beachten. Nicht alle Operatoren binden ihre Operanden gleich stark.

Hier ein paar der wichtigsten Operatoren, geordnet und gruppiert nach ihrer Präzedenz. Je weiter unten der Operator steht, desto geringer seine Präzedenz.

Operator	Bedeutung
~	Bitweises NICHT
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
«	Bitweises shiften
&	Bitweises UND
^	Bitweises XOR
|	Bitweises ODER

4.2 Zuweisungsoperatoren

Mit den Zuweisungsoperatoren weist man den Variablen neue Werte zu. Dabei unterschiedet man zwischen direkter und kombinierter Zuweisung.

Bei einer direkten Zuweisung übergibt man direkt einen neuen (berechneten) Wert einer Variable:

```
a = 42;
a = b + c;
```

Wenn man aber mit dem Wert einer Variable »a« etwas berechnen möchte und das Ergebnis wieder auf »a« ablegen möchte, bieten sich die kombinierte Operatoren an.

Man kann jeden arithmetischen Operator mit einem Gleichheitszeichen verbinden und man hat einen kombinierten Operator, der nach dem folgenden Schema funktioniert:

```
a = a + b;  -->  a += b;
a = a * b;  -->  a *= b;
a = a | b;  -->  a |= b;
```

Auch eine Verkettung mit mehreren Operatoren und Variablen ist denkbar:

```
a = b = c;
```

Es gibt außer den obigen Operatoren noch den Dekrement und den Inkrement. Mit ihnen ist es möglich den Wert einer Variable zu erhöhen oder zu verringern. Dabei gibt es zwei Varianten: Präfix und Postfix.

Der Präfix-Operator hat die zweithöchste Präzedenz und wird somit vor fast allen anderen Operatoren ausgeführt. Deshalb ist es möglich einen Wert vor einer Zuweisung, einem Vergleich oder einer Berechnung zu verändern.

```
a = 1;
b = 3;
b = ++a; //b=2
```

Der Postfixoperator hat im Vergleich zum Präfix eine geringere Präzedenz und wird somit nach einer Zuweisung, einem Vergleich etc. ausgeführt.

```
a = 1;
b = 3;
b = a++; //b=1
```

4.3 Vergleichsoperatoren

Vergleichsoperatoren werden in Kontrollstrukturen, wie beispielsweise *if*-Bedingungen benötigt. Auch hier sind Präzedenzen zu beachten!

Operator	Bedeutung
<	Kleiner
>	Größer
<=	Kleiner gleich
>=	Größer gleich
==	Gleichheit
!=	Ungleichheit

Hier ein Beispiel für einen einfachen Vergleich:

```
if (a == b)
{
   ...
}
```

4.4 Aussagenlogik

Mit Hilfe der Aussagenlogik lassen sich beispielsweise Bedingungen verknüpfen oder negieren. Sie sind sehr hilfreich, wenn man mehrere Bedingungen mit *und* bzw. *oder* verknüpfen will. Dadurch werden komplexere Kontrollstrukturen erst möglich gemacht.

Operator	Bedeutung
!	Logisches NICHT
&&	Logisches UND
||	Logisches ODER

In dem folgenden Beispiel ist die Bedingung erst erfüllt, wenn »a« gleich »b« oder nicht-»a« gleich »c« entspricht.

```
if ((a == b) || (!a == c))
{
   ...
}
```

5 Funktionen

Eine Funktion erfüllt den Zweck, redundanten Code zu vermeiden und Struktur in ein Programm zu bringen. Eine Funktionalität, welche mehrfach benötigt wird, z.B. das Auslesen einer Zeile von einer Datei, sollte man als Funktion implementieren.

Dabei hat eine Funktion immer einen *Ausgabewert*, *Ausgabetyp*, *Namen*, *Parameter* und *Parametertypen*.

Die folgende Funktion heißt »getResult« und gibt eine ganze Zahl aus. Sie benötigt für die Berechnung zwei ganze Zahlen.

```
int getResult(int a, int b)
{
  return (--a + b);
}
```

Dabei gibt es in C auch die Möglichkeit, Prozeduren zu deklarieren. Der Rückgabetyp dafür ist »void«.

Falls man eine Funktion konstruiert, die keine Parameter benötigt, schreibt man entweder nichts in die Klammer oder »void«.

```
void noReturn(void)
{
  printf("I'm in space!");
  printf("SPACE!!!");
}
```

5.1 Einstiegsfunktion

Eine besondere Form der Funktion ist die Einstiegsfunktion, oder auch *main()*-Funktion.

Es ist die erste Funktion, die direkt beim Start aufgerufen wird und bildet somit den Einstiegspunkt eines Programms.

Außerdem werden die Parameter, mit dem wir ein Programm starten, an die Einstiegsfunktion übergeben. Dafür gibt es zwei Variablen: Parameterzähler und Parameter-Array.

Der Parameterzähler beinhaltet die Anzahl an Parametern, die übergeben worden sind, wobei sie niemals kleiner als eins wird. Der erste Parameter ist nämlich immer der Programmname selbst, da er auch als Parameter zählt.

Das Parameter-Array ist quasi eine »Liste« mit allen Parametern. Da es sich um einen char-Pointer-Array handelt, kann man direkt auf die Zeichenketten via Arrayindex zugreifen.

Aufruf im Terminal:

```
$./program.x foobar
```

Programmcode:

```
int main(int argc, char *argv [])
{
  printf("%s", argv[0]); //Output: ./program.x
  printf("%s", argv[1]); //Output: foobar
  exit(EXIT_SUCCESS);
}
```

6 Header

Header sind eigenständige Dateien, welche Funktionsprototypen und Definitionen von Variablen oder Konstanten beinhalten können.

Dadurch, dass man eine Funktion im Header beschreibt, kann man diese Funktion in anderen Codedateien benutzen, indem man den entsprechenden Header einbindet. Die Implementation der Funktion befindet sich in einer anderen Codedatei.

Somit kann man mit Headern Programme modular aufbauen, was die Übersichtlichkeit und Struktur enorm verbessern kann.

Dabei gibt es zwei verschiedene Arten Header einzubinden: Systemheader und eigene Header. Die Systemheader liegen bei GNU/Linux-Systemen zumeist unter »/usr/include/*«. Die eigenen Header werden entweder absolut oder relativ verlinkt.

```
#include <stdlib.h>     //Systemheader
#include "src/demo.h"  //Eigener Header
```

6.1 Beispiel

In dem Header »demo.h« befindet sich eine Variablendefinition und ein Funktionsprototyp. Beide sind nicht implementiert oder deklariert!

```
extern int foo;
int bar(int);
```

Die Deklaration/Implementation steht in der Codedatei »demo.c«:

```
#include "demo.h"

int foo = 42;

int bar(int a)
{
  ...
}
```

6.2 Standardheader

Eine Liste mit Standardheadern, die man fast immer braucht:

- *stdlib.h*: Eine Standardbibliothek mit vielen nützlichen Funktionen
- *stdio.h*: Bietet Funktionen für Ein-/Ausgabe an
- *string.h*: Liefert viele Funktionen zur Verarbeitung von Zeichenketten
- *stdbool.h*: Definiert den Datentyp »bool«

7 I/O – Schnittstelle zu einem Programm

Die Ein- und Augabe ermöglicht es dem Benutzer mit einem Programm interaktiv zu arbeiten. Dabei reden wir hier von der Ein-/Ausgabe in einem Standardterminal.

7.1 Ausgabe

Es ist weit verbreitet, dass man zwei verschiedene Ausgabeströme besitzt. Zum einen den Standardausgabestrom (stdout) und des Weiteren einen Fehlerausgabestrom (stderr).

In C gibt es die Funktion »fprintf()«, mit der man Zeichenketten ausgeben kann. Dabei gibt man bei dieser Funktion zwei Parameter an: Ausgabestrom und Zeichenkette.

```
fprintf(stdout, "Hello␣World!");     //Standardausgabe
fprintf(stderr, "Goodbye␣World!"); //Fehlerausgabe
```

Die Funktion »printf()« dagegen gibt nur auf der Standardausgabe aus. Wenn man normale Zeichenketten ausgeben will, sollte man diese Funktion benutzen, da es den Code lesbarer gestaltet.

7.2 Eingabe

Die Eingabe, die man via Terminal übergibt, kann man mit der Funktion »scanf()« auslesen. Als Parameter übergibt man einen Formatstring und einen Pointer. Der ausgelesene Inhalt wird dann in die entsprechende Variable im Speicher geladen.

Dabei ist zu beachten, dass diese Funktion so lange liest und auf den Speicher schreibt, wie es der Formatstring vorgibt. D.h. wenn man ein begrenztes *char*-Array deklariert hat und *scanf()* keine Limitierung vorsetzt, überschreibt er den Speicher (Pufferüberlauf). Das ist gefährlich, da dadurch Code überschrieben werden kann und somit ein Einstiegspunkt für Hacker bietet.

Man kann dies verhindern, indem man in dem Formatstring eine begrenzte Anzahl an Symbolen akzeptiert.

```
char foo[42];
scanf("%42s", &foo);
```

8 Neuerungen in C11

C11 ist der neu-verabschiedete Standard der ISO. Wie jeder neue C-Standard baut es auf den alten Standards auf und erweitert sie. Zu den wichtigsten neuen Features gehören:

- Standardbibliothek für Multithreading
- Bessere Unterstützung von Unicode
- Transparente Structs
- Exklusiver Dateizugriff

Thread-Unterstützung Der C11 besitzt nun endlich eine eigene Implementation für das Multithreading, wobei sie an »PThreads« angelehnt ist. Durch die Implementation im Standard ist es nun möglich, Multithreading systemunspezifisch anzubieten.

Transparente Structs Sie selbst sind namenlose Structs und man verwendet sie, wenn man Structs verschachteln möchte. Innerhalb eines Structs sind somit keine Bezeichner mehr für die Structs nötig, was meistens sowieso unsinnig erscheint.

Exklusiver Dateizugriff Beim Öffnen/Erstellen einer Datei besteht immer die Frage, ob sie schon existiert. Deswegen gibt es den exklusiven Zugriff: Wenn eine Datei schon existiert, dann wird sie eingelesen; falls sie jedoch nciht existiert, wird sie neu erstellt und dann eingelesen.

Compiler Das Problem bei neuen Sprachstandards ist immer, dass sie sich eine Zeit lang etablieren müssen, bis die größeren Compiler (clang, gcc, etc.) alle Features des neuen Standards implementiert haben. Bis jetzt gibt es leider nur eine rudimentäre Unterstützung.

9 Referenzen

http://www.heise.de/developer/artikel/C11-Neue-Version-der-Programmiersprache-
Teil-1-1661014.html http://www.open-std.org/jtc1/sc22/wg14/www/standards http://upload.wikimedia.org/wik
754-single.svg http://de.wikipedia.org/wiki/C_%28Programmiersprache http://upload.wikimedia.org/wikipedi
http://www.smart-v2g.info/images/logos/iec-logo.svg http://de.wikipedia.org/wiki/C_%28Programmiersprach
http://en.wikipedia.org/wiki/C99#Design http://en.wikipedia.org/wiki/C11_%28C_standard_revision%29
http://de.wikibooks.org/wiki/C-Programmierung:_Ausdr%C3%BCcke_und_Operatoren