Ausarbeitung zum Thema:

Sprachstandards von C

von
David Weiser

Proseminar: C-Programmierung – Grundlagen und Konzepte

Inhaltsverzeichnis

Vorwort

In der vorliegenden Arbeit werde ich Ihnen den Sprachstandard der Programmiersprache C vorstellen und erklären. Begonnen wird mit einer kurzen Einführung über C, in der einige grundlegende Eigenschaften und geschichtliche Hintergründe von C erläutert werden. Anschließend werde ich die Entwicklung bis hin zum aktuellen Sprachstandard C99 skizzieren und diesen detailliert vorstellen.
Zum Abschluss gebe ich dann noch einen kurzen Ausblick auf den neuen Sprachstandard (C1X) und werde schon bekannte Unterschiede von C99 und C1X vorstellen.

1. Einführung

Die Programmiersprache C zählt zu den imperativen Programmiersprachen. Das bedeutet, dass genau angegeben werden muss, WIE etwas berechnet wird. Ein Programm einer imperativen Programmiersprache besteht aus Abfolgen von Anweisungen, die ein Rechner in einer definierten Reihenfolge abarbeitet und so den Inhalt von Speicherstellen verändert.

C wurde von 1971-1973 von David Ritchie gemeinsam mit den Bell Laboratories entwickelt. Die eigentliche Motivation war es eine Sprache zu entwickeln, die für die Systemprogrammierung für das Betriebssystem Unix verwendet werden konnte und als Sprachabstraktion zur Assemblersprache diente. Somit waren direkte Speicherzugriffe und sehr hardwarenahe Programmierung möglich. Noch immer sind bzw. werden viele Betriebssystem-Kernels in C geschrieben, was zu einer hohen Verbreitung von C und den dafür nötigen Compilern führt/führte.

Der C-Sprachstandard (C Standard Library) bietet eine Reihe von Standard-Funktionen an, die vom Compiler implementiert und mit neuen Revisionen des Sprachstandards fortwährend erweitert werden. Dazu gehören unter anderem Funktionen zum Manipulieren von Zeichenketten, für die Typkonvertierung oder für datei- oder konsolenbasierte Ein- und Ausgabe. In der Theorie sollte also ein, in einem jeweiligen Sprachstandard programmiertes C-Programm, von jedem, den Sprachstandard implementierenden Compiler, übersetzt und ausgeführt werden können. Da die Compilerhersteller jedoch weitere Bibliotheken implementieren können, ist dies oftmals nicht der Fall. Für den heute aktuellen C99 Standard gibt es bis heute keine vollständige Implementierung.

2. Der Weg zum Sprachstandard C99

Nachdem 1973 der ersten Unix-Kernel in C geschrieben werden konnte, veröffentlichten Brian Kerningham und Dennis Ritchie 1978 das Buch „The C Programming Language“. Diese Version von C wird nach ihren Erfindern auch „K&R C“ genannt und diente, da es noch keine Standardisierung der Sprache gab, als informelle Referenz.

Die erste „offizielle“ C-Version wurde 1989 vom ANSI-Komitee unter dem Namen C89 herausgegeben, welche 1990 schließlich unter dem Namen C90 mit einigen Änderungen auch von der ISO übernommen wurde. Die von Kerningham und Ritchie in ihrem Buch beschriebenen Spracheigenschaften wurden darin normiert. Außerdem wurden weitere Sprachelemente hinzugefügt und der Umfang der Standardbibliothek festgelegt. Zu den neu hinzugefügten Sprachelementen zählen z.B. die Möglichkeit der Prototypen-Deklaration, Konstanten-Deklaration aber auch ein verbesserter Präprozessor, welcher Anweisungen zum Einschleusen von Quelltext bearbeitet (siehe Bild 2.1).

1995 gab die ISO die Sprachversion C95 heraus. Diese enthielt neben Fehlerkorrekturen auch Änderungen am Sprachumfang. Es wurden Standard-Makros eingeführt, so dass nun z.B. „and“ statt „&&“ geschrieben werden konnte oder das „STDC_Version“-Makro (siehe Bild 2.1). Mit letzterer Präprozessoranweisung konnte überprüft werden in welchem C-Standard (C90, C95 oder später C99) das Programm oder ein Teil davon geschrieben wurde. So war es möglich Teile des Quelltextes nur unter bestimmten C-Standards auszuführen bzw. Programme für mehrere C-Standards kompatibel zu machen.

1999 veröffentlichte die ISO schließlich die Sprachversion C99. Neben den üblichen Fehlerkorrekturen wurden aus C++ bekannte Sprachelemente aufgenommen. Dazu gehört unter anderem der mit *//* eingeführte Zeilenkommentar, das Schlüsselwort *inline* oder die Möglichkeit der Variablendeklaration innerhalb einer for-Schleife. Weitere Neuheiten waren die Unterstützung von komplexen Zahlen und die Erweiterung ganzzahliger Datentypen.
Der Sprachstandard C99 wird heute auch informell C0X genannt nachdem drei *Technical Corrigenda* einflossen.

```
#if defined(__STDC_Version__) && __STDC_Version__ >= XX
Quelltext
#endif
```

(Bild 2.1; Präprozessoranweisung zum Überprüfen des C-Standards)

Auf Bild 2.1 ist eine Präprozessoranweisung zu sehen. Diese Anweisungen sind daran zu erkennen, dass sie mit einem *#schlüsselwort* eingeleitet und abgeschlossen werden. Im vorliegenden Fall mit *#if* und *#endif* womit dann, mit Hilfe des Makros *__STDC_VERSION__*, der Sprachstandard geprüft wird unter dem der *Quelltext* ausgeführt werden soll.
Die Sprachstandards haben die folgenden „Versionsnummern“, welche dann für **XX** eingesetzt werden können:

- C89/99: ---
- C95: 199409L
- C99: 199901L

3. Der Sprachstandard C99

1. Variablen und Konstanten

C stellt, wie jede andere Programmiersprache, eine Reihe von Basis-Datentypen für die Deklaration von Variablen und Konstanten zur Verfügung. Da sich die Deklaration von Konstanten etwas von der der Variablen unterscheidet, wird hierauf noch einmal speziell eingegangen. Die Menge der Basis-Datentypen wurde bei der Entwicklung der einzelnen Sprachstandards ständig erweitert. Auf Neuerungen im C99 Standard wird zusätzlich hingewiesen.

Eine Besonderheit von C ist die Möglichkeit Zahlen als *signed* (vorzeichenbehaftet) oder *unsigned* (vorzeichenlos) zu deklarieren, wodurch sich deren Wertebereich verändert. Während als *signed* deklarierte Variablen auch negative Werte im Wertebereich zulassen, haben als *unsigned* deklarierte Variablen nur einen positiven Wertebereich. Standardmäßig sind alle Variablen *signed*.

Zeichen und Zahlen

Begonnen wird mit dem Datentyp *char* (=character). Dieser Datentyp hat eine vorgegebene Größe von einem Byte, was auf den gewöhnlichen Rechnerarchitekturen 8 Bit entspricht. Es ergibt sich ein *unsigned*-Wertebereich von 0 bis 255 und einen *signed*-Wertebereich von -127 bis +127, womit sich kleine Ganzzahlen darstellen und verarbeiten lassen. Die „Hauptaufgabe“ des Datentyp *char* ist allerdings die Darstellung der 256 Zeichen aus der ASCII-Tabelle. Jede Zahl von 0 bis 255 codiert dabei ein ASCII-Zeichen (Beispiel siehe Bild 3.1).

```c
// Hier werden über die Präprozessoranweisung #include weitere Bibliotheken eingebunden
#include <stdlib.h>
#include <stdio.h>

// Die main-Funktion. Stellt den Einstiegspunkt für ein C-Programm dar.
// Sie gibt standardmäßig ein int zurück und
// bekommt keinen Parameter (deshalb void) übergeben.
int main(void)
{
    // Deklaration von zwei char Variablen mit dem Namen firstChar und secondChar
    char firstChar;
    char secondChar;

    // Initialisierung der beiden char Variablen

    // firstChar wird direkt mit einem ASCII-Zeichen initialisiert
    firstChar = 'A';

    // secondChar bekommt eine Zahl zugewiesen,
    // die aber das ASCII-Zeichen 'B' repräsentiert
    secondChar = 66;

    // Mit den folgenden zwei Zeilen werden die beiden chars auf der Konsole ausgegeben
    printf("Der erste Buchstabe: %c \n", firstChar);
    printf("Der zweite Buchstabe: %c", secondChar);


    // Rückgabe eines int. Für unseren Fall uninteressant.
    return 0;
}
```

(Bild 3.1; Kommentiertes Beispiel zur Benutzung des Datentyps *char*)

Ausgabe des Beispiels von Bild 3.1:
Der erste Buchstabe: A
Der zweite Buchstabe: B

Neben dem Datentyp *char* gibt es noch die Integer-Datentypen. Im Gegensatz zu einem *char*, welcher auch aufgrund seines kleinen Wertebereichs vorrangig dazu verwendet wird die Zeichen des ASCII-Alphabets darzustellen und eher weniger für „größere Rechnungen“ genutzt wird, haben die Integer-Datentypen einen größeren Wertebereich.

In Tabelle 3.1 sind einige Integer-Datentypen mit ihrer Byteanzahl (1 Byte = 8 Bit) und ihrem Wertebereich aufgeführt. Der Datentyp *long* bezeichnet den Datentyp mit der natürlichen Größe. Seine Größe passt sich dem jeweiligen System an. Auf einem 32-Bit-System hat er 4 Byte (32 Bit), auf einem 64-Bit-System eben 8 Byte (64 Bit). Zu beachten ist noch, dass manche C-Systeme einen negativen Wertebereich mit z.B. -32.768 (*short*), -2.147.483.648 (*long*) oder auch -128 (*char*) zulassen. Dies ist davon abhängig ob die 0 intern zweimal (-0 und +0) dargestellt wird.

Datentyp	Größe	Wertebereich *signed*	Wertebereich *unsigned*
short	2 Byte	-32.767 - +32.767	0 – 65.535
int	4 Byte	-2.147.483.647 - +2.147.483.647	0 - 4.294.967.295
long	Natürliche Größe (z.B. 4 oder 8 Byte)	z.B. -2.147.483.647 - +2.147.483.647 oder -9.223.372.036.854.755.808 - +9.223.372.036.854.755.808	z.B. 0 – 4.294.967.295 oder 0 - 18.446.744.073.709.551.615
long long*	8 Byte	-9.223.372.036.854.755.808 - +9.223.372.036.854.755.808	0 - 18.446.744.073.709.551.615
int_least8_t*	Mind. 8 Bit (1 Byte)	Mind. -127 - +128	Mind. 0 - 255
int32_t*	Exakt 32 Bit (4 Byte)	-2.147.483.647 - +2.147.483.647	0 - 4.294.967.295

(Tabelle 3.1; Größe und Wertebereiche einiger Integer-Datentypen; 1Byte = 8 Bit; die mit * markierten Einträge sind Neuerungen, die erst seit dem C99-Standard mit in die Sprache eingeflossen sind)

Neben den Integer-Datentypen für ganze Zahlen gibt es Datentypen für die Darstellung von Fließkommazahlen (Tabelle 3.2). Im Sprachstandard C99 sind dies die Datentypen *float, double* und *long double.* Im Gegensatz zu den Integer-Datentypen können Fließkommazahlen auch Kommazahlen darstellen. Diese Eigenschaft ermöglicht genauere mathematische Berechnungen. Im Gegensatz zu den ganzen Zahlen sind Fließkommazahlen immer *signed*.

Ein C-Standard legt nicht fest wie die Fließkommazahlen intern dargestellt werden. Jedoch wird vorrangig der IEEE-Standard 754 verwendet. Der Wertebereich kann also je nach Implementierung unterschiedlich sein. Über die Headerdatei *float.h* kann der genaue Wertebereich ermittelt werden.

Datentyp	Größe	Wertebereich
float (einfache Genauigkeit)	4 Byte	1.2E-38 bis 3.4E+38
double (doppelte Genauigkeit)	8 Byte	2.3E-308 bis 1.7E+308
long double*	10 Byte	3.4E-4932 bis 1.1E+4932

(Tabelle 3.2; Größe und Wertebereich der Fließkommadatentypen; 1 Byte = 8 Bit; die mit * markierten Einträge sind Neuerungen, die erst seit dem C99-Standard mit in die Sprache eingeflossen sind)

Konstanten

Konstanten sind Variablen, die ihre Werte nicht mehr ändern. In C gibt es zwei Möglichkeiten Konstanten zu deklarieren.
Bei der ersten Möglichkeit (siehe Bild 3.2) wird die Konstante einfach durch die Präprozessoranweisung *#define*, gefolgt vom Konstantennamen und -wert, deklariert. Diese Art der Deklaration bewirkt ein einfaches „Suchen und Ersetzen“ durch den Compiler. An jeder Stelle im Quelltext, an der der Konstantenname auftritt (hier zahl1), wird er durch den vorher angegebenen Wert ersetzt (hier 20).

```
#define zahl1 20
```

(Bild 3.2; Konstantendeklaration und Initialisierung mit *#define*)

Eine andere Art Konstanten zu deklarieren ist auf Bild 3.3 zu sehen. Hier erfolgt die Deklaration durch das Schlüsselwort *const*. Es wird vom Datentyp, Namen und Wert der Konstanten gefolgt. Der Vorteil zur vorherigen Variante ist die Überprüfung von Datentyp und Wertebereich der Konstanten durch den Compiler.

```
const int zahl2 = 21
```

(Bild 3.3; Konstantendeklaration und Initialisierung mit *const*)

Variablenverhalten

Der C Standard bietet einige Schlüsselwörter bei der Deklaration von Variablen um deren Verhalten festzulegen. Zum einen gibt es das Schlüsselwort *static*, welches die Variable an einer festen Speicheradresse speichert. Die Variable wird dabei vom Compiler initialisiert und immer wenn anschließend auf die Variable zugegriffen wird, liest sie ihren alten Wert aus der festgelegten Speicheradresse. Das Beispiel aus Bild 3.4 hat die Konsolenausgabe:

Erste Ausgabe: 4
Zweite Ausgabe: 5

Obwohl die Funktion *test()* zweimal nacheinander aufgerufen wird, wird die Variable *number* nur einmal initialisiert. Beim zweiten Aufruf der Funktion wird an der Stelle der Initialisierung der Variablen dann auf die vorher festgelegte Speicheradresse und auf die durch den ersten Funktionsaufruf geänderte Variable zugegriffen.

```c
// Hier werden über die Präprozessoranweisung #include weitere Bibliotheken eingebunden
#include <stdlib.h>
#include <stdio.h>

// Die main-Funktion. Stellt den Einstiegspunkt für ein C-Programm dar.
// Sie gibt standardmäßig ein int zurück und
// bekommt keinen Parameter (deshalb void) übergeben.
int main(void)
{
    // In zwei verschiedenen printf()-Funktionen wird jeweils einmal
    // test() aufgerufen, welche einen int zurück gibt.
    printf("Erste Ausgabe: %d \n", test());
    printf("Zweite Ausgabe: %d", test());

    // Rückgabe eines int. Für unseren Fall uninteressant.
    return 0;
}

// Eine Funktion mit dem Namen test und dem Rückgabewert int
int test(void)
{
    // Deklaration und Initialisierung einer static Variablen
    // vom Typ int, dem Namen number und dem Wert 3.
    static int number = 3;

    // Der Wert der Variablen number wird um eins erhöht.
    number = number + 1;

    // Rückgabe von number
    return number;
}
```

(Bild 3.4; Beispiel für die Verwendung von *static*-Variablen)

Eine weitere Möglichkeit das Verhalten von Variablen zu bestimmen ist das Schlüsselwort *volatile*. Dieses Schlüsselwort kann benutzt werden um dem Compiler zu zeigen, dass die Variable außerhalb des Programmkontextes geändert wird. Dies ist z.B. dann der Fall wenn eine Variable über das System durch Nutzereingaben verändert wird.

Mit dem Schlüsselwort *register* werden Variablen gekennzeichnet, die durch den Compiler direkt in das Register geschrieben werden sollen. Das ist meist dann von Vorteil, wenn eine Variable oft benutzt wird und man sich den schnelleren Speicherzugriff auf Register zu Nutze machen will (beispielsweise in einer Schleife). Heutige Compiler schreiben Variablen bei Bedarf aber automatisch in Register, weswegen dieses Schlüsselwort auch weggelassen werden kann.

2. *Typumwandlung*

Wie in anderen Programmiersprachen ist es auch in C möglich den Typ einer Variablen im Quelltext durch Typumwandlung zu verändern. Man unterscheidet dabei die explizite und implizite Typumwandlung.

Die implizite Typumwandlung wird durch den Compiler durchgeführt. In dem Beispiel in Bild 3.5 ist dies in Zeile 19 der Fall. Die *int*-Variable *inumber* mit dem Wert 65 wird der *char*-Variablen *sign* zugewiesen. *Sign* hat nun den Wert 65, was dem Zeichen 'A' entspricht.

Die explizite Typumwandlung ist auf Bild 3.5 in Zeile 22 zu sehen und muss im Quelltext angegeben werden (hier durch *(int)*). Die *float*-Variable *fnumber* mit dem Wert 12.123 wird der *int*-Variablen *inumber2* zugewiesen. Als Ausgabe erhält man aber nicht 12.123 sondern 12. Das liegt daran, dass der Datentyp *int* nur ganze Zahlen darstellen kann. Die Nachkommastellen werden bei der expliziten Typumwandlung also einfach abgeschnitten. Man nennt sie deswegen auch *einschränkende Typumwandlung*.

```
1  // Hier werden über die Präprozessoranweisung #include weitere Bibliotheken eingebunden
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  // Die main-Funktion. Stellt den Einstiegspunkt für ein C-Programm dar.
6  // Sie gibt standardmäßig ein int zurück und
7  // bekommt keinen Parameter (deshalb void) übergeben.
8  int main(void)
9  {
10     // Deklaration und Initialisierung einer int und einer float Variable.
11     int inumber = 65;
12     float fnumber = 12.123;
13
14     // Deklaration einer char und einer int Variable.
15     char sign;
16     int inumber2;
17
18     // Eine implizite Typumwandlung.
19     sign = inumber;
20
21     // Eine explizite Typumwandlung.
22     inumber2 = (int) fnumber;
23
24     // Konsolenausgabe
25     printf("Erste Ausgabe: %c \n", sign);
26     printf("Zweite Ausgabe: %d", inumber2);
27
28     // Rückgabe eines int. Für unseren Fall uninteressant.
29     return 0;
30 }
```

(Bild 3.5; Beispiel zur Typumwandlung)

Ausgabe:
Erste Ausgabe: A
Zweite Ausgabe: 12

C bietet mit den Funktionen *scanf()* und *printf()* zwei Funktionen für die Ein- bzw. Ausgabe auf der Konsole an. Diese gehören jedoch nicht zum eigentlichen Sprachumfang und müssen erst von einem sogenannten Linker eingebunden werden.

Auf Bild 3.6 sind einige *printf()*-Funktionsaufrufe zu sehen. Als ersten Parameter bekommt die Funktion einen *String* übergeben den sie auf die Konsole ausgeben soll. In dem *String* sind durch % eingeleitete Formatelemente enthalten, die der Reihe nach die durch Kommata getrennten Werte in den *String* einbauen. In unserem Beispiel ist dies pro *printf()*-Funktion immer nur ein Wert. Durch die Formatelement lassen sich die eingelesenen Werte jedoch auch direkt umwandeln.

Das Beispiel aus Bild 3.6 hat die Ausgabe:
Zahl 20: 24
Zahl 20: +20
Zahl 20: 00020
Zahl 20.543586: 20.544

```
printf("Zahl 20: %o", 20);
printf("\nZahl 20: %+i", 20);
printf("\nZahl 20: %05d", 20);
printf("\nZahl 20.543586: %10.3f", 20.543586);
```

(Bild 3.6; Beispiele für *printf()*-Funktionsaufrufe)

Ausgewählte Formatzeichen sind in Tabelle 3.3 erklärt. Der Buchstabe nach dem %-Zeichen gibt dabei den Zieltyp an.

Formatzeichen	Ausgabe des Wertes als...
%d oder %i	*int*
%o	Oktalzahl
%x	Hexadezimal
%f	*float*
%c	char

(Tabelle 3.3; Formatzeichen)

Zusätzlich zu den Formatzeichen kann das %-Zeichen mit weiteren Angaben zur Umwandlung des Wertes versehen werden. Direkt nach dem %-Zeichen werden die *Flags* (Tabelle 3.4) angefügt, dann die Angaben zur Feldbreite, die Anzahl der Nachkommastellen und schließlich das eigentliche Formatzeichen. Die Anteil der Nachkommastellen an der Feldbreite wird durch einen Punkt getrennt angegeben. In der letzten Zeile unsere Beispiels in Bild 3.6 hat die Ausgabe der Zahl also insgesamt 10 Stellen von denen aber drei für die Angabe der Nachkommastellen benutzt werden.

Symbol	Funktion
-	Text wird links ausgerichtet
+	Vorzeichen einer Zahl wird immer mit ausgegeben
Leerzeichen	Positive Zahlen beginnen mit einem Leerzeichen
0	Felder werden mit Nullen aufgefüllt anstatt mit Leerzeichen
#	Hat verschiedene Bedeutungen, je nachdem mit welchem Formatzeichen es zusammen verwendet wird. %#x liest z.B. die Zahl als Hex-Zahl ein und stellt ihr ein 0x vor

(Tabelle 3.4)

Die *scanf()*-Funktion (Bild 3.7) liest hingegen Werte ein und speichert sie in einer Variablen ab. Sie erwartet allerdings nicht den Namen der Variablen (hier *x*), sondern deren Adresse. Für diesen Fall stellt man dem Variablennamen einfach den Adressoperator *&* vor. Wie bei der *printf()*-Funktion kann man auch bei der *scanf()*-Funktion Formatelemente angeben und den Wert direkt in das gewünschte Format umzuwandeln.
Analog zu den Formatzeichen der *prinf()*-Funktion wandeln %i und %d den Wert in einen *int*-Wert um, während %o den Wert als Oktalzahl und %x als Hexadezimalzahl einlesen.

```
scanf("%d", &x);
```

(Bild 3.7; Beispiel für *scanf()*-Funktionsaufruf)

4. Operatoren

Die Standard-C-Funktionen umfassen auch noch eine Reihe von Operatoren. Dazu gehören die Prä- und Postfix-Operatoren, die Shift-Operatoren und die Logik-Operatoren.

Die Prä- bzw. Postfix-Operatoren bestehen aus den Inkrement(++)- und Dekrement(--)-Operatoren. Mit diesen ist es möglich eine Zahl um eins zu erhöhen bzw. vermindern. *Prä-* und *postfix* bezeichnen die Stellung des Operators in Verbindung mit der Variable. In Bild 3.8 wird die Anwendung der Operatoren verdeutlicht.
In beiden Fällen soll die Variable *x* mit dem Wert 20 um eins erhöht werden. Bei der Postfix-Version *x++* in der zweiten Zeile wird der Variable *zahl1* der Wert 20 zugewiesen und anschließend um eins erhöht. Bei der Präfix-Variante ++x in Zeile drei wird der Wert der Variable *x* zunächst um eins erhöht und erst dann erfolgt die Zuweisung an die Variable *zahl2.* Im zweiten Fall wurde also auch der Wert der Variable *x* um eins erhöht.

```
int x = 20;
int zahl1 = x++;
int zahl2 = ++x;
```

(Bild 3.8; Beispiel für die *Prä-* und *Postfix*-Operatoren)

Zu den Shift-Operatoren zählen << und >>. Diese Operatoren arbeiten auf der Bit-Darstellung der Werte. Sie schieben die Bits eines Wertes um einen Bit nach links bzw. rechts. << entspricht einer Multiplikation mit 2, >> einer Division durch 2.

Die Logik-Operatoren bestehen unter anderem aus den Vergleichsoperatoren < (kleiner), > (größer), >= (größer gleich), <=(kleiner gleich), *!=* (ungleich) und == (gleich). Außerdem gehören *&&,* || und *!* dazu. Der Rückgabetyp der Vergleichsoperatoren ist ein *int*-Wert. 0 steht dabei für *false*, jede andere Zahl steht für *true.* Erst im C99-Standard wurde ein eigener Datentyp für die boolschen Werte *true* und *false* geschaffen.

5. Kontrollstrukturen

Kontrollstrukturen sind auch in C ein wichtiges Sprachelement. Um bedingte Anweisungen auszuführen, gibt es die *if*-Anweisung und die *switch*-Anweisung.
Außerdem gibt es Schleifen, die es möglich machen einen Teil des Quelltexts mehrere Male auszuführen. Dazu gehören in C die *for-, while-* und *do-while*-Schleife. Eine *for-each*-Schleife, die auf jedes Element einer Menge eine bestimmte Funktion anwendet, gibt es in C hingegen nicht.

In dem Quelltextausschnitt in Bild 3.9 ist eine *if*-Anweisung zu sehen. Welche Funktion hier ausgeführt wird, hängt von dem Wert von *x* ab. In Zeile 12 wird zunächst überprüft, ob *x* den Wert 2 hat. Trifft dies zu, wird auf der Konsole der Text *x ist 2* ausgegeben. Wenn *x* nicht den Wert 2 besitzt, wird mit *else if* überprüft ob *x* den Wert 3 hat. Ist *x* nun 3, wird *x ist 3* auf der Konsole ausgegeben. In allen anderen Fällen greift der *else*-Zweig der *if*-Anweisung und es wird *x ist weder 2 noch 3* ausgegeben.

```
10      int x;
11
12      if (x == 2)
13      {
14          printf("x ist 2");
15      }
16      else if (x == 3)
17      {
18          printf("x ist 3");
19      }
20      else
21      {
22          printf("x ist weder 2 noch 3");
23      }
```
(Bild 3.9; Beispiel einer *if*-Funktion)

Auf Bild 3.10 sieht man eine *switch*-Anweisung. Sie hat eigentlich die gleiche Funktion, wie die *if*-Anweisung aus Bild 3.9. Eine *switch*-Funktion wird meistens dann verwendet, wenn es mehrere Fälle zur Überprüfung gibt, da eine *if*-Anweisung dann schnell unübersichtlich wird.
Die *switch*-Anweisung beginnt dabei mit dem Schlüsselwort *switch* gefolgt von dem Variablennamen, dessen Wer überprüft werden soll (in unserem Fall *x*). Anschließend werden mit dem Schlüsselwort *case* die Werte, auf die *x* überprüft werden soll, aufgeführt (hier 2 und 3). Jeder *case* hat dann eine Funktion, die ausgeführt werden soll, wenn *x* dem im jeweiligen *case* angegebenen Wert entspricht. Jeder *case* hat außerdem noch eine *break*-Anweisung. Dies bewirkt ein Abbrechen der *switch*-Anweisung sobald eine Bedingung erfüllt wurde (die anderen Bedingungen werden dann nicht mehr überprüft). Am Ende des Beispiels in Bild 3.10 ist noch das Schlüsselwort *default* zu finden. Sollte keine Bedingung zutreffen, wird immer diese *default*-Anweisung ausgeführt.
Die Ausgabe der Funktion wäre analog zur obigen *if*-Funktion. Hat *x* den Wert 2, wird *x ist 2* ausgegeben. Hat *x* den Wert 3, wird *x ist 3* ausgegeben. In allen anderen Fällen wird *x ist weder 2 noch 3* ausgegeben.

```
10      int x;
11
12      switch (x)
13      {
14      case 2:
15          printf("x ist 2");
16          break;
17
18      case 3:
19          printf("x ist 3");
20          break;
21
22      default:
23          printf("x ist weder 2 noch 3");
24          break;
25      }
```
(Bild 3.10; Beispiel einer *switch*-Anweisung)

In Bild 3.11 ist eine *for*-Schleife zu sehen, die den Quelltext zwischen den geschweiften Klammern (Zeile 12 und 14) so oft ausführt, wie es die Abbruchbedingung der Schleife zulässt.
In Zeile 10 wird zunächst die Zählervariable *i* deklariert. In Zeile 11 beginnt die *for*-Schleife mit dem Schlüsselwort *for*. Die Werte in der Klammer sind folgendermaßen zu verstehen: Die Zählervariable mit Namen *i* wird zunächst auf den Wert 1 gesetzt. Getrennt durch ein Semikolon wird nun die Abbruchbedingung der Schleife angegeben. So lange unsere Zählervariable *kleiner oder gleich* 5 ist, wird die Schleife durchlaufen. Anschließend wird, wieder durch ein Semikolon getrennt, angegeben, was nach einem Schleifendurchlauf mit Zählervariable geschehen soll. In unserem Fall wird sie nach jedem Durchlauf um eins erhöht. Erreicht sie dann schließlich den Wert 6 wird die Schleife abgebrochen.

```
10      int i;
11      for (i = 1; i <= 5; i++)
12      {
13          printf("Die %d. Runde\n", i);
14      }
```

(Bild 3.11; Beispiel einer *for*-Schleife)

Die Ausgabe der Beispiele aus Bild 3.11, 3.12 und 3.13 wäre:
Die 1. Runde
Die 2. Runde
Die 3. Runde
Die 4. Runde
Die 5. Runde

Zwei weitere Schleifen, die von C zur Verfügung gestellt wird, sind die *While*- und *do-while*-Schleifen (Bild 3.12 und 3.13). Beide funktionieren recht ähnlich. Bei beiden wird zunächst die Zählervariable initialisiert. Im Falle der *while*-Schleife folgt nun das Schlüsselwort *while* gefolgt von der Abbruchbedingung. In den geschweiften Klammern steht nun wieder der Quelltext, der so lange ausgeführt werden soll, so lange die Zählervariable nicht die Abbruchbedingung verletzt (also so lange sie nicht größer als 5 ist).
Die *do-while*-Schleife beginnt mit dem Schlüsselwort *do.* Anschließend kommen die geschweiften Klammern mit dem Quelltext, der ausgeführt werden soll und am Ende, nach dem Schlüsselwort *while,* folgt die Überprüfung der Abbruchbedingung.

Während in der *while*-Schleife die Abbruchbedingung am Anfang überprüft wird, wird sie in der *do-while*-Schleife erst am Ende überprüft. In Bild 3.13 wird also, egal ob *i* kleiner oder gleich 5 ist, die *printf()*-Funktion auf jeden Fall einmal ausgeführt.
Im Gegensatz zur *for*-Schleife muss die Zählevariable auch innerhalb des Blocks mit den geschweiften Klammern bearbeitet werden (in unserem Fall wird sie durch *i++* um eins erhöht).

```
10      int i = 1;
11      while (i <= 5)
12      {
13          printf("Die %d. Runde\n", i);
14          i++;
15      }
```

(Bild 3.12; *while*-Schleife)

```
10      int i = 1;
11      do
12      {
13          printf("Die %d. Runde\n", i);
14          i++;
15      }
16      while (i <= 5);
```

(Bild 3.13; *do-while*-Schleife)

6. Funktionen

Wie wir schon in den vorherigen Beispielen gesehen haben, arbeitet C mit Funktionsaufrufen. In z.B. Java werden die Funktionen *Methoden* genannt. Sie bilden quasi eine Quelltext-Einheit, die nur als Ganzes ausgeführt werden kann, die eine gewisse Funktionalität bereit stellt und die innerhalb des Programms beliebig oft aufgerufen werden kann. Ziel von Funktionen ist es, gerade bei größeren Programmen, den Quelltext zu strukturieren.

Eine Funktion hat in C die in Bild 3.14 dargestellte Form. Der Rückgabetyp beschreibt dabei den Datentyp des Wertes, der durch das immer ganz am Ende stehende *return*-Statement zurückgegeben wird, wenn die Funktion beendet wird. Außerdem hat die Funktion einen Namen, mit dessen Hilfe sie an anderen Stellen des Quelltexts aufgerufen werden kann. In den Klammern hinter dem Namen stehen die Parameter. Dies sind Werte, die der Funktion beim Aufruf übergeben werden müssen. Zwischen den geschweiften Klammern stehen die Statements, die beim Aufruf der Funktion ausgeführt werden.

```
Rueckgabetyp Name(Parameter)
{
    statement;
    return-statement;
}
```

(Bild 3.14; Aufbau einer Funktion)

Eine weitere Möglichkeit um in C Funktionen zu deklarieren besteht in der Nutzung von Funktionsprototypen. Ein Beispiel für die Form dieser Prototypen ist in Bild 3.15 zu sehen. Sie bestehen nur aus dem Rückgabetyp, dem Namen und den evtl. zu übergebenen Parametern. Es ist darauf zu achten, dass es sich hier nur um eine Funktionsdeklaration handelt und die Funktion noch keine Funktionalität implementiert hat. Die Funktionalität muss an anderer Stelle im Quelltext definiert werden. Dazu muss der Rückgabetyp, Name und die Parameter bei der Definition identisch zu denen in der Deklaration (also denen des Prototypen) sein.

```
Rueckgabetyp Name(Parameter);
```

(Bild 3.15; Muster Funktionsprototyp)

C bietet außerdem noch die Möglichkeit Funktionen als *inline* zu deklarieren. *Inline*-Funktionen bieten einen „schnellstmöglichen Funktionsaufruf“. Der Compiler kopiert dabei den Funktionsinhalt an die Stelle des Funktionsaufrufs. Somit entfällt im *Aufrufstack* der Sprung zur Funktion und der Rücksprung. Viele Compiler ignorieren aber das Schlüsselwort *inline* und entscheiden selbst, wann eine Funktion als *inline* ausgeführt werden soll.
Das Muster einer Deklaration einer *inline*-Funktion ist auf Bild 3.16 zu sehen. Es unterscheidet sich, bis auf das eingefügte Schlüsselwort *inline* nicht von einer normalen Funktionsdeklaration.

```
inline Rueckgabetyp Name(Parameter)
{
    statement;
    return-statement;
}
```

(Bild 3.16; Muster einer *inline*-Funktion)

7. Header

Mit *Headern* bietet C die Möglichkeit sein Programm zu modularisieren und Quelltext wiederzuverwenden. Möchte man also eine Funktion oder auch Konstanten aus einem anderen Programm verwenden, kann man diese in einen *Header* kapseln und sie über die Präprozessoranweisung *#include* in sein Programm einbinden. Innerhalb seines Programms kann man dann auf die in dem *Header* deklarierten Funktionen und Konstanten zugreifen, als wären sie Teil des Programms selbst. Eine Headerdatei besteht dabei immer aus dem Headernamen gefolgt von einem *.h*.

8. Arrays

Eines der wichtigsten Sprachelemente sind *Arrays*. Sie bieten die Möglichkeit mehrere Werte eines Datentyps in einer Liste zu speichern. Mit Hilfe der vorgestellten Schleifen, ist es dann möglich durch die *Arrays* zu iterieren und Funktionen auf die einzelnen Elemente anzuwenden.
Es gibt in C ein- und zweidimensionale *Arrays*. Auf Bild 3.17 ist die Deklaration eines ein- und eines zweidimensionalen *Arrays* zu sehen. Bei beiden *Arrays* muss zunächst der Typ der Element angegeben werden, die in dem *Array* gespeichert werden sollen (hier int). Das eindimensionale *Array myArray* bekommt über sein Parameter *x* in den eckigen Klammern die Größe des Arrays übergeben. Es kann dann *x* Element des Typs *int* speichern. Der Zugriff auf die Elemente des *Arrays* erfolgt dann über den Index beginnend bei 0. Um auf das erste Element eines *Arrays* zuzugreifen, muss also auf den Index 0 zugegriffen werden.

Das zweidimensionale *Array myArray2* bekommt hingegen zwei Parameter *x* und *y* übergeben. Die interne Darstellung kann man sich dann als Tabelle vorstellen, wobei *x* die Zeilen- und y die Spaltenanzahl angibt. Auch hier beginnen die Indizies mit 0. Um also auf das erste Element des *Arrays* zuzugreifen muss man auf Zeile = 0 und Spalte = 0 zugreifen.

In Bild 3.18 ist die Initialisierung zweier *Arrays* zu sehen. Das eindimensionale *Array myArray* besteht aus drei *int*-Elementen. Die Elemente können nach dem =-Zeichen direkt in eckigen Klammern angegeben werden. In unserem Fall also 1, 2 und 3.
Das zweidimensionale *Array myArray2* besteht aus zwei Zeilen und drei Spalten, wobei auch diese Werte direkt in eckigen Klammern angegeben werden können. Die Einträge des *Arrays* werden dabei zeilenweise in den eckigen Klammern angegeben. Die erste Zeile (mit Index 0) besteht also aus den Zahlen 1, 2 und 3 während die zweite Zeile (Index 1) die Einträge 4, 5 und 6 hat.
Eine andere Möglichkeit der Initialisierung eines zweidimensionalen *Arrays* kann bei *myArray3* beobachtet werden. Während hier nur angegeben wird, dass das *Array* zwei Zeilen und drei Spalten haben soll, werden die Elemente ohne Aufteilung auf die Zeilen angegeben, sondern einfach nur nacheinander in eckigen Klammern angegeben. C wird diese Element aber automatisch auf die zwei Zeilen und drei Spalten aufteilen, wodurch *myArray3* genauso aufgebaut ist wie *myArray2*.

Wird bei der Initialisierung keine Elementanzahl in der eckigen Klammer angegeben, wird das *Array* automatisch so groß, dass es alle angegebenen Elemente speichern kann.

```
int myArray[x];
int myArray2[x][y];
```

(Bild 3.17; Deklaration von Arrays)

```
int myArray[3] = {1, 2, 3};

int myArray2[2][3] = {
                        {1,2,3}
                        {4,5,6}
                    };

int myArray3[2][3] = {1,2,3,4,5,6};
```

(Bild 3.18; Initialisierung von Arrays)

Seit dem Sprachstandard C99 ist es auch möglich *Arrays variabler Größe* zu deklarieren. Bei diesen *Arrays* hängt die Größe von einer Variablen ab, die zur Laufzeit des Programms geändert werden kann. In unserem Beispiel würde dies heißen, dass im Fall von *myArray* eine Variable *x* deklariert (als *int*) und mit einem beliebigen Wert initialisiert wird. Da eine Funktion diese Variable nun während des Programmablaufs ändern kann, würde sich auch die Größe des *Arrays* ändern. Je nachdem, welchen Wert die Variable *x* bei der Initialisierung des Arrays hat. Dies ist z.B. dann nötig, wenn die Anzahl der Elemente, die im *Array* gespeichert werden sollen bei der Compilierung noch nicht feststehen.

9. Strings

In C gibt es, im Gegensatz zu z.B. Java, keinen Datentyp für *Strings*. *Strings* sind in C *Arrays* vom Datentyp *char.* In Bild 3.19 ist die Initialisierung zweier *char-Arrays* zu sehen, die jeweils einen *String* repräsentieren.
Im ersten Fall, also bei der Initialisierung des *Arrays string1,* werden in den geschweiften Klammern nacheinander die einzelnen *char* aus denen der *String* bzw. das *Array* bestehen soll angegeben. Zu beachten ist, dass bei dieser Art der Initialisierung das *Stringende*-Zeichen *\0* angegeben werden muss. Das *Array string1* hat also 5 Elemente (Index 0 – 4).
Eine zweite Möglichkeit einen *String* zu erstellen zeigt die Initialisierung des *Arrays string2*. Hier wird dem *Array* vom Typ *char* in der eckigen Klammer einfach das Wort, welches der *String* repräsentieren soll, angegeben. Das *Stringende*-Zeichen muss in diesem Fall nicht mit angegeben werden. Dies geschieht automatisch.
Trotz der unterschiedlichen Art der Initialisierung repräsentieren *string1* und *string2* den gleichen *String*, nämlich *test*.

```
char string1[] = { 't', 'e', 's', 't', '\0'};
char string2[] = {"test"};
```

(Bild 3.19; Initialisierung zweier *Strings*)

4. Ausblick

Der Sprachstandard C wird stetig weiter entwickelt. Genauso wie 1995 C95 als Weiterentwicklung von C89/C90 veröffentlicht wurde, ist C99 eine Weiterentwicklung von C95. So ist zum Zeitpunkt des Verfassens dieser Arbeit ein Komitee mit der Weiterentwicklung des C99-Standards betraut. Arbeitstitel dieser Weiterentwicklung ist C1X. Im folgenden sind nun einige schon bekannte Änderungen des C1X-Standards aufgeführt.

- Neue Datentypen *char16_t* und *char32_t* zur Unterstützung von UTF-16 bzw. UTF-32 UniQuelltext-Zeichen.

- Unterstützung von Multithreading.
  Über den *Header <threads.h>* werden Funktionen zum Erstellen und Verwalten von Threads bereitgestellt.
  Über den *Header <stdatomic.h>* werden Funktionen zum Ausführen von atomaren, also unteilbaren, Aktionen bereitgestellt.

- Prüfung von *Array*-Grenzen zur Laufzeit.

- Unterstützung der dezimalen Darstellung von Fließkommazahlen.
  Der Standard ist IEEE-754-2008.

- Prüfung der Speicherausrichtung von Objekten.
  Speicherausrichtung kann über den *Header <stdalign.h>* ermittelt werden.

5. Literaturverzeichnis

- Varianten der Programmiersprache C
http://de.wikipedia.org/wiki/Varianten_der_Programmiersprache_C

- C (Programmiersprache)
http://de.wikipedia.org/wiki/C_(Programmiersprache)

- C-Programmierung
http://de.wikibooks.org/wiki/C

- Das C-Tutorial
http://www.c-howto.de/tutorial.html

- Programmiersprache C/C++
http://www.imb-jena.de/~gmueller/kurse/c_c++/index.html

- Galileo Computing – C von A bis Z
http://openbook.galileocomputing.de/c_von_a_bis_z/

- C1X
http://en.wikipedia.org/wiki/C1X