C

Klaus Knopper

Ein wenig Geschichte

1970 Ken Thomson entwickelt auf einer PDP/7 die Sprache B

1972 Entwicklung von C durch Dennis Ritchie in Bell Telephone Laboratories

1978 Standardwerk „The C programming language“ von Kernighan

1979 Stroustrup entwickelt „C with Classes“, später C++

1980 ANSI-Standardisierung

1991 ISO-Standardisierung

1994 Ergänzungen zum ANSI-Standard (ISO C Amendment 1)

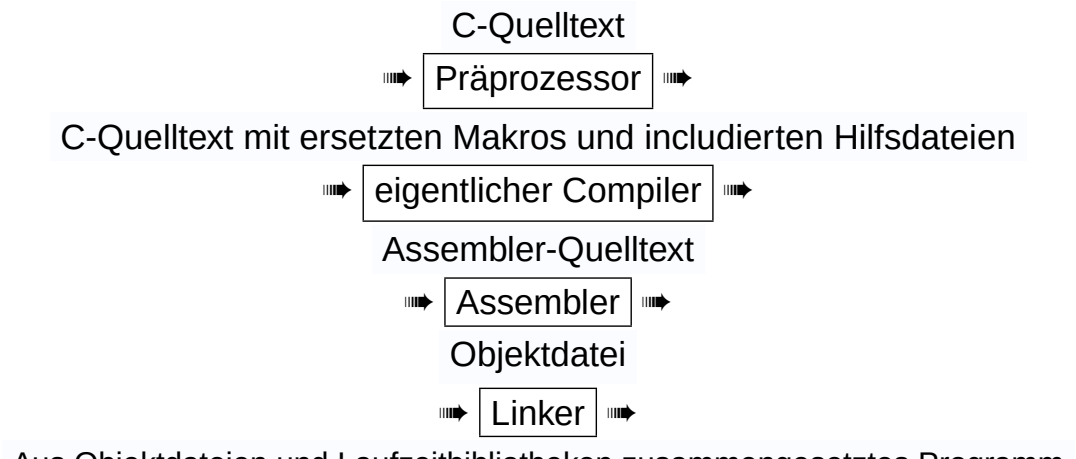
Programmiersprachen

sind *formale Sprachen* zum Schreiben und Darstellen von Computerprogrammen. Sie dienen dazu, reale Probleme mit Hilfe von Maschinen zu lösen, bzw. der Maschine das Problem und den Lösungsansatz
einzuprägen.

Durch einen Text mit logischen Befehlen, dem Quellcode, werden dabei dem Computersystem die internen Verarbeitungsschritte vermittelt.

Compiler

Der Compiler übersetzt „natürlichsprachliche“ Anweisungen, die das Problem und die Lösung abstrahieren, in von der Maschine ausführbaren Code. Dies geschieht im Fall von C in 4 Schritten:



Linker und Bibliotheken

Die **Objektdateien** enthalten architekturspezifischen, binären Maschinencode, der aus den Quelltexten erzeugt wurde. Mehrere Objektdateien können in Form von **Bibliotheken** zusammengefasst werden, die Funktionen und Prozeduren zur Verfügung stellen, für die eine Deklaration in den Header-Dateien existiert, so dass sie von den selbst programmierten Programmteilen mitverwendet werden können. Ein Beispiel hierfür ist die C(++)-Standardbibliothek, die von fast jedem Programm mitverwendet wird.

Der **Linker** bindet sogenannte Startup-Objektdateien mit den aus den Quelltexten erzeugten Objektdateien und Bibliotheken zu ausführbaren Programmen.

Beispiel: Der GNU C Compiler **gcc**

Erstellen der Objektdatei `testprog.o` für `testprog.c`

```
gcc -c testprog.c
```

Linken der Objektdatei `testprog.o`

```
gcc testprog.o
```

Kompilieren und Linken in einem Schritt

```
gcc testprog.c
```

Fehlt die Angabe einer Ausgabedatei, so wird das ausführbare Programm in die Datei **`a.out`** gespeichert. Mit der Option `-o` kann eine Ausgabedatei angegeben werden:

```
gcc -o testprog testprog.c
```

Entwicklungsumgebungen

...sind i.d.R. graphische Benutzerinterfaces (GUIs), die einen Editor mit farblicher Hervorhebung von bekannten C/C++-Strukturen (Syntaxhighlighting) und Buttons zur Übersetzung, Ausführung und schrittweiser Ausführung unter Darstellung der Inhalte von Variablen (Debugging) enthalten. Auch Vorlagen für komplexe Programmstrukturen, Dokumentation und Konfigurationshilfen für verschiedene Rechnerplattformen (`Makefiles`, `configure`-Skripte) sind oft Bestandteil von Entwicklungsumgebungen.
Generell ist es natürlich auch ohne Entwicklungsumgebung möglich, Programme zu entwickeln, solange für das entsprechende Betriebssystem ein Compiler und ein einfacher Texteditor vorhanden ist. Auch textorientierte Hilfprogramme bieten einen gewissen Komfort (`make`, `autoconf`/`configure`, `m4`).
Beispiele für graphische Entwicklungsumgebungen: **Eclipse**, **kdevelop**, **KDE Studio** (Linux/Unix), *Visual C++* (MS-Windows).

Vordefinierte Datentypen

```
int ganzzahl;
char zeichen;
float fließkommezahl;
double doppeltgenaue_fließkommazahl;
```

Varianten von **`int`**

```
short int ganzzahl;
long int ganzzahl;
long long int ganzzahl;
unsigned int ganzzahl;
signed int ganzzahl;
const int ganzzahl;
```

Achtung: Derzeit sind long long (64-Bit auf Intel) Integervariablen zwar Bestandteil des C-ISO (c99) Standards, aber noch nicht im C++-ISO Standard enthalten. Viele Compiler (z.B. der GNU C++ Compiler) unterstützen diesen Datentyp zwar, Sie sollten sich jedoch nicht darauf verlassen, dass ihn jeder Compiler beherrscht.

Die Funktion **`sizeof()`**

...bestimmt die Größe (in Bytes = 8 bit) eines Datentyps. Die Größen einiger Datentypen wie double können sich je nach Rechnerplattform unterscheiden.

```
#include <stdio.h>
int main()
 {
  printf("Ein char belegt %d Bytes\n",
            sizeof(char) );
  printf("Ein int belegt %d Bytes\n",
            sizeof(int) );
  printf("Ein unsigned int belegt %d Bytes\n",
            sizeof(unsigned int) );
  printf("Ein double belegt %d Bytes\n",
            sizeof(double) );
  return 0;
 }
```

Grundgerüst

Beim Start des ausführbaren Programms wird die Funktion `main(int argc, char *argv[])` aufgerufen. `argv[]` ist ein Datenfeld mit Zeichenketten, die dem Programm auf der Eingabekommandozeile übergeben werden können, `argc` enthält die Anzahl der übergebenen Zeichenketten einschließlich des Namens der ausführbaren Datei. `int main()` kann einen Fehlerwert an das aufrufende Programm oder das Betriebssystem übergeben, oder den Wert `0`, um die erfolgreiche Ausführung anzuzeigen.

```
#include <stdio.h> // für Ein-/Ausgabe
int main(int argc, char *argv[])
 {
  printf("Hallo, Welt!\n");
  return 0;
 }
```


Kommentare

... sind geeignet, die einzelnen Konstrukte eines Quelltextes näher zu erläutern. Jedes Programm sollte vom Autor durch Kommentare dokumentiert werden, um die Verständlichkeit und Nachvollziehbarkeit zu erhöhen. Kommentare können überall im Programm stehen.

Ein Kommentar steht zwischen der Zeichenfolge **`/*`** und **`*/`**.

```
 /* Dies
    ist
    ein Kommentar */
```

Mit C++ wurde die Konstruktion

```
// Kommentar bis zum Ende der Zeile...
```

als Alternative für Kommentare eingeführt, was im Falls des GCC Compilers auch für C-Programme erlaubt ist, da der gleiche Präprozessor wie bei C++ verwendet wird.

Kommentare, Beispiel:

```
#include<stdio.h>
int main() {
float f = 40.0F; /* Temperatur in Grad Fahrenheit */
float c = 0.0F;  /* Temperatur in Grad Celsius */
/* Grad in Celsius umrechnen */
c = ((f - 32.0F ) * 5.0F ) / 9.0F;
/* Temperatur in Celsius und Fahrenheit ausgeben */
printf("%.2g Grad Fahrenheit entspricht", f);
printf("%.2g Grad Celsius\n", c); // 2 Kommastellen
}
```


Variablen

... sind Speicherstellen für Daten. Jede Variable hat einen *Namen* und einen zugehörigen *Datentyp*. Vor der Verwendung einer Variable muss sie *deklariert* werden. Numerische Variablen wie **`int`** sind, sofern sie nicht schon bei der Deklaration mit einem Wert initialisert werden, unbestimmt.

```
int x = 2;
short y = 30/4;
long z = x*27;
```

Regeln für Variablennamen

Variablennamen dürfen i.d.R. nur aus Zahlen, Buchstaben und einem Unterstrich (_) bestehen. Bestimmte Sonderzeichen wie Umlaute und Symbole, die keine C-Schlüsselwörter sind, sind zwar bei vielen Compilern erlaubt, aber nicht in jedem Fall vom verwendeten Zeichensatz unabhängig, und sollten daher vermieden werden.
Das erste Zeichen einer Variablen muss ein Buchstabe oder ein Unterstrich sein. Schlüsselwörter wie double sind ungültig.
Variablen-Namen sollten möglichst aussagekrätig sein.

```
char Taste;
int zaehler5 = 0;
double lange_zahl;
```


Konstanten

... sind spezielle Variablen, denen einer fester Wert zugewiesen wird. Eine spätere Modifikation des Variablenwerts ist unzulässig. Ansonsten werden Konstanten wie einfache Variablen behandelt.

```
const int antwort = 42;
const double steuer_satz = 0.08;
```

Operatoren

... sind Aktionen, die auf einen oder mehrere Operanden angewendet werden. In C sind alle Operanden Ausdrücke, die von den Operatoren verarbeitet werden.

Es gibt vier verschiedene Arten von Operatoren:

- Zuweisungsoperatoren
- mathematische Operatoren
- Vergleichsoperatoren
- logische Operatoren

Zuweisungsoperatoren

Der Zuweisungsoperator ist das Zeichen **=** und unterscheidet sich von der Schreibweise in der Mathematik. Mittels Zuweisungsoperator wird einer Variable ein Wert zugewiesen, welcher Ergebnis eines Ausdrucks ist.

Die Syntax ist: **`variable = ausdruck;`**

Mathematische Operatoren

In C gibt es 2 unäre und 5 binäre Operatoren. Zu den unären Operatoren gehören der Inkrementoperator **++** und der Dekrementoperator **--**.

```
#include <stdio.h>
int main(){
int x,y;
int i=5;
x = ++i; // i wird um 1 erhöht und x zugewiesen
y = i++; // i wird y zugewiesen und danach um 1 erhöht
printf("x = %d\n",x); // 6
printf("y = %d\n",y); // 6
printf("i = %d\n",i); // 7
}
```


Vergleichsoperatoren

Ein Vergleichsoperator vergleicht Ausdrücke. Meist wird dieser Operator im Zusammenhang mit bedingten Anweisungen benutzt.
Es gibt die 6 Vergleichsoperatoren:

- **==** (gleich)
- **<=** (kleiner gleich)
- **>=** (größer gleich)
- **<** (kleiner)
- **>** (größer)
- **!=** (ungleich)

Ist das Ergebnis eines Ausdrucks ungleich 0, so entspricht dies dem Vergleichergebnis *wahr*, ist das Ergebnis 0 so bedeutet dies *falsch*.

Logische Operatoren

Die logischen Operatoren sind **||** (logisches ODER), **&&** (logisches UND) und ! (NOT). Logische Operatoren werden meist in zusammengesetzten Bedingungen eingesetzt.
Dabei gelten folgende Regeln:

```
(ausdruck1) && (ausdruck2)
```

ist wahr, wenn **ausdruck1** und **ausdruck2** beide wahr sind

```
(ausdruck1) || (ausdruck2)
```

ist wahr, wenn mindestens einer der beiden Ausdrücke wahr ist

```
(!ausdruck)
```

ist dann wahr, wenn **ausdruck** falsch ist

Priorität und Assoziativität (1)

Jeder Operator hat eine bestimmte **Priorität**. Der Operator mit der höheren Priorität wird in einem Ausdruck zuerst ausgewertet.

```
-4 + 4 * 5 + 3
```

ergibt 19, da ***** eine höhere Priorität als **+** hat.

Sind in einem Ausdruck mehrere Operatoren der gleichen Priorität vorhanden, so wird von links nach rechts (**Assoziativität**) ausgewertet.

```
12 % 5 * 2 // % = MODULO liefert den Teilungsrest
```

ergibt 4, da zuerst **12 % 5** ausgewertet wird.

Priorität und Assoziativität (2)

Durch Setzen von Klammern können Operatoren eine höhere Priorität erhalten und somit wird der entsprechende Ausdruck zuerst ausgewertet.

```
(14 + 9) % (1+2)
```

ergibt 23 % 3 = 2

Bibliotheksfunktionen: **`printf()`**

Die Funktion **`printf()`** ist nicht Teil der Sprache C selbst, sondern eine Funktion aus der C-Standardbibliothek.
`printf()` hat folgende Signatur:

```
printf("Format-String",variablen ...);
```

Der Format-String kann normalen Text zur Ausgabe, aber auch Format-Anweisungen enthalten, z.B.:

`%d`	Ausgabe einer `int`-Ganzzahl
`%ld`	Ausgabe einer `long int`-Ganzzahl
`%s`	Ausgabe einer `char*`-Zeichenkette
`%f`	Ausgabe einer `double`-Fließkommazahl
`%g`	„platzsparende“ Ausgabe einer `double`-Fließkommazahl

Die für die jeweilige Format-Anweisung zuständigen Variablen werden, durch Komma getrennt, an den Format-String angehängt.

Beispiel für **`printf()`**

```
#include <stdio.h> // für Ein-/Ausgabe
#include <math.h>  // für sin()
int main(int argc, char *argv[])
 {
  printf("%s(%d) = %f\n", "Sinus", 111, sin(111.0) );
  return 0;
 }
```


Bibliotheksfunktionen: **`scanf()`**

`scanf()` hat folgende Signatur:

```
int scanf("Format-String",Adressen von Variablen
...);
```

`scanf()` ist das Gegenstück zu **`printf()`**, und kann Variablen, deren *Speicheradresse* in der Parameterliste hinter dem Format-String übergeben werden, durch Tastatureingaben belegen. Der Format-String ist im Wesentlichen mit dem von **`printf()`** identisch. Vorsicht ist geboten, wenn ein Variablentyp einen begrenzten Speicherbereich belegt, in der Eingabe jedoch viel mehr Daten übergeben werden. Dies passiert z.B. bei Zeichenketten sehr leicht, daher ist hier darauf zu achten, dass nur bestimmte Längen von Eingabewerten akzeptiert werden.

Als Rückgabewert liefert **`scanf()`** eine Ganzzahl mit der Anzahl erfolgreich eingelesener Variablenwerte zurück.

Beispiel für **`scanf()`**

```
#include <stdio.h> // für Ein-/Ausgabe
int main(int argc, char *argv[])
 {
  int eingabe;
  int anzahl = scanf("%d", &eingabe);
  return 0;
 }
```

Hinweis: **`&eingabe`** bedeutet „die Speicheradresse der Variable **`eingabe`**“. Dies ist notwendig, damit **`scanf()`** weiß, wohin es die von Tastatur eingelesenen Daten kopieren soll. (vergl. *Call by Value* vs. *Call by Reference*).

Bedingte Ausführung (**`if ... else`**)

```
if( a==b )
 {
   printf("a und b sind gleich!\n";
 }
else
 {
   printf("a und b sind nicht gleich!\n");
 }
```

Mit dem **ternären Operator** geht es kürzer:

```
printf("a und b sind ");
printf( a==b ? "gleich" : "verschieden" );
printf("\n");
```

Mehrfachvergleiche - **`switch()`**

Eine Möglichkeit, wiederholte **`else if ...`**-Konstrukte abzukürzen, bietet **`switch()`**:

```
switch(c)
 {
   case ’q’: antwort=0; break;
   case ’j’:
   case ’y’: antwort=1; break;
   default:  printf("Keine Antwort angegeben.\n");
  }
```

Bedeutung: Wenn die Variable **`c`** ein q enthält, wird **`antwort`** auf 0 gesetzt, falls **`c`** ein **`j`** oder **`y`** enthält, wird **`antwort`** auf 1 gesetzt, ansonsten wird „Keine Antwort angegeben“ ausgegeben.
Etwas gewöhnungsbedürftig: Erst das **`break;`** innerhalb der **`switch()`**-Umgebung beendet die Anweisungen, die ab der zutreffenden Bedingung ausgeführt werden sollen!

`break; continue;`

`break`- und **`continue`**-Anweisungen sind nur im Rumpf von Schleifen und in der switch-Anweisung zulässig. **`break`** bricht dabei die Ausführung der Schleife bzw. der **`switch`**-Anweisung sofort ab und *verlässt* diese. Eine **`continue`**-Anweisung bricht ebenfalls den aktuellen Durchlauf ab, springt aber zum nächsten Durchlauf der Schleife.

```
for(;;) // Endlosschleife
 {
  ++zaehler;
  if(zaehler > 10) break; // Abbrechen
  if(zaehler % 2) continue; // Rest überspringen
  printf("%d\n", zaehler);
 }
```

Programmende mit **`exit()`**

In der Regel ist das Programmende erreicht, wenn die letzte Anweisung in der **`main()`**-Funktion abgearbeitet ist, oder **`main()`** mit **`return`** verlassen wird.

Mit der exit() Funktion ist eine vorzeitige Programmebeendigung, auch aus Funktionen außerhalb von **`main()`** heraus, möglich.

Der Funktion **`exit()`** werden dabei sog. „exit-Codes“ übergeben, die ggf. vom Betriebsystem ausgewertet und weiterverarbeitet werden können.

```
exit(0); // erfolgreiches Programmende
exit(1); // Programmausführung war fehlerhaft
```


Präprozessor - Anweisungen

Mit „#“ beginnende Zeilen kennzeichnen Präprozessor-Anweisungen, und sind kein eigentlicher Bestandteil der Programmiersprache C (oder auch C++)! Prinzipiell lassen sich auch gewöhnliche Texte (Dokumentationen, Briefe, Bücher) mit Hilfe von Präprozessoranweisungen neu zusammenfügen und verändern. Es gibt sogar richtige Programme, die ausschließlich mit Hilfe des Präprozessors als „Interpreter“ funktionieren, und die durch Definitionen und Makros, die dem Präprozessor i.d.R. auf der Kommandozeile übergeben werden, gesteuert werden.

```
#if defined(__STDC__)
# warning Dies ist ein ANSI-konformer C/C++-Compiler.
# warning Herzlichen Glückwunsch.
#else
# error Dieses Programm benötigt einen ANSI-Compiler!
#endif
```

Präprozessor - **#include <datei>**

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

fügt den Inhalt der Datei **stdio.h** aus dem voreingestellten Suchpfad des Präprozessors in das aktuelle Programm ein, bevor der eigentliche C-Compiler aufgerufen wird. Prinzipiell können Sie auch eigene Dateien **#include**n, hierbei verwenden Sie allerdings

```
#include "meinefunktionen.h"
```

anstelle der < ... >, damit auch der für lokale Dateien zusätzliche Suchpfad durchsucht wird, und Sie Ihre Include-Datei nicht in irgendwelchen Systemverzeichnissen fest installieren müssen.

Präprozessor - Makros (1)

Konstante Ausdrücke, dies sind z.B. Hilfe-Texte oder unveränderliche Zahlenkonstanten, können nach einem einfachen Suche/Ersetze-Schema vom Präprozessor in Form von leicht zu merkenden symbolischen „Alias“en verwaltet werden.

Der Präprozessor tauscht die selbstdefinierten Symbole im Programmquelltext aus, bevor der eigentliche Compiler das Programm übersetzt.

```
#define M_PI 3.14159265358979323846 // math.h

...

w = cos(2.0 * M_PI * phi);
```

Präprozessor - Makros (2)

Es ist Konvention, Präprozessor-Makros in GROßBUCHSTABEN zu schreiben, um sie von C/C++-Variablen und Funktionen leichter unterscheiden zu können.

Makros können ähnlich wie Funktionen in C aufgebaut sein, denn der Präprozessor unterstützt auch variable Ersetzungsargumente in Makro-Aufrufen.

```
#define QUADRAT(x) ((x)*(x))
...
double seitenlaenge=2.0, flaeche;
flaeche = QUADRAT(seitenlaenge);
```

Überlegen Sie, warum und in welchen Fällen die hier etwas umständlich aussehende Klammerung der Argumente in der Makro-Definition wichtig sein könnte!

Funktionen - Deklaration

Funktionen müssen in C vor ihrer ersten Verwendung zumindest **deklariert** werden (d.h. der Compiler muss über Syntax, Typ des Rückgabewertes und Typ und Anzahl der verwendeten Parameter informiert werden).

```
[extern] double quadrat(double argument);
```

Der Compiler fügt an dieser Stelle noch keinen Code ein, sondern erlaubt ab der Deklaration einer Funktion deren syntaktische Prüfung im Programm. Andernfalls wird mit dem Fehler "**`undeclared function`**" abgebrochen. Mit dem optionalen Stichwort **extern** wird gekennzeichnet, dass die eigentliche Implementation einer Funktion außerhalb der aktuellen Quelltextdatei erfolgt.

Funktionen - Deklarationen in Hilfsdateien

Während sich der eigentliche Code von vordefinierten Funktionen in den C-Standardbibliotheken befindet, muß Ihr Programm dennoch durch entsprechende Deklarationen erfahren, wie eine Funktion aufgerufen wird und was sie an Parametern erwartet. Hierfür sind die **`include`-Dateien** verantwortlich.

```
#include <math.h> // Mathematische Funktionen
                  // aus math.h importieren
```


Hilfsdateien

In Hilfsdateien werden i.d.R. Funktionen, Variablen und Klassen (bei C++) nur **deklariert**, nicht jedoch **definiert** bzw. **implementiert**. Es ist nicht nur guter Stil, sondern zur Übersichtlichkeit und zur Vermeidung von Fehlern wie z.B. unterschiedliche Mehrfachdeklaration erforderlich, alle in mehreren **`.c`**-Dateien (d.h. separaten Teilen eines größeren Programms) verwendeten Funktionen und Variablen in von diesen Programmteilen gemeinsam includierten Hilfsdateien zu deklarieren.

Funktionen - Definition

Funktionen müssen, damit sie aufgerufen werden können, natürlich auch inhaltlich **definiert** bzw. **implementiert** werden.

```
double quadrat(double argument)
 {
  // Ergebnis als double an den Aufrufer
  // von quadrat(x) zurückgeben
  return argument * argument;
 }
...
double flaeche;
flaeche = quadrat(seitenlaenge);
```

Eine solche Funktions-**Implementation** innerhalb einer C-Quelltextdatei hat implizit auch eine Funktions**deklaration** zur Folge (falls dies nicht schon früher geschehen ist).

Rekursive Funktionen

Unter einer rekursiven Funktion versteht man eine Funktion, die sich selbst aufruft. Dies ist z.B. bei rekursiv definierten mathematischen Funktionen sinnvoll, hierdurch wird auch oft der Quelltext übersichtlicher.

```
int fakultaet(int argument)
 {
  if(argument <= 1) return 1;
  return argument * fakultaet(argument - 1);
 }
```

Bei rekursiven Funktionen muss unbedingt darauf geachtet werden, dass bei allen Eingaben irgendwann im Laufe der Rekursion das Abbruchkriterium erreicht wird, da die Funktion (und möglicherweise das komplette Programm) sonst nicht terminiert.

while-Schleife

```
int zaehler=1;
while(zaehler <= 10) // Solange zaehler <= 10
 { // Beginn while-Schleife
 printf("%d\n",zaehler++); // Ausgeben und hochzählen
 } // Ende while-Schleife
```


for-Schleife

```
// Initialisierer (einmalig);
// Bedingungsabfrage (vor Durchlauf);
// Kommando (nach Durchlauf)

 int zaehler;
 for(zaehler=0; zaehler<10; zaehler++)
  {
    printf("Counter: %d", zaehler);
  }
```

do...while-Schleife

```
int zaehler=0;
do
 { /* Beginn while-Schleife */
  printf("%d", zaehler++); /* Ausgeben und hochzählen */
 } // Ende while-Schleife
while(zaehler < 10); /* Semikolon nicht vergessen! */
```

Die Schleife wird mindestens einmal durchlaufen, die Überprüfung der Bedingung erfolgt **nach** jedem Durchlauf.

Arrays (Felder)

Ein Feld ist ein Sammlung von Datenelementen, die alle den gleichen Namen und den gleichen Datentyp besitzen. Die Elemente sind dabei linear geordnet und der Zugriff erfolgt über Indizes.

```
int feld[10]; /* Feld mit 10 int-Werten
                 von feld[0] bis feld[9] */

int i;
for(i=0; i<10; i++) // von 0...9
 {
  feld[i] = i*i; // Quadrat berechnen und zuweisen
 }
```

Mehrdimensionale Arrays

... besitzen mehrere Indizes, über die auf die Feldelemente zugegriffen werden kann.

```
#include <stdio.h>
int main()
{
 int feld[2][2] = {4, 6, 9, 2}, a, b;

 for(a=0; a<2; a++)
   for(b=0; b<2; b++)
     printf("feld[%d][%d] = %d\n", a, b, feld[a][b]);
 return 0;
}
```

	feld[*][0]	**feld[*][1]**
feld[0][*]	4	6
feld[1][*]	9	2

struct (Verbund)

... ist eine Sammlung von mehreren Variablen unter einem Namen, über die auf die einzelnen Elemente zugegriffen werden kann. Die Variablen können dabei verschiedenen Datentypen angehören.

```
struct Buch
 {
  int katalognummmer;
  char autor[80];
  char titel[80];
  char kurzbeschreibung[128];
 }; // Semikolon nach Deklaration nicht vergessen
...

struct Buch b = { 0, "Unbekannt", "C",
                  "Über die Tücken des Objekts"};
b.katalognummer = 12345;
```

Verschachtelte Verbünde

`struct`s können andere **`struct`**s enthalten.

```
struct koord{
    int x;
    int y;
};

struct quadrat {
   struct koord obenlinks;
   struct koord untenrechts;
 } quad1;
```

Auf die Elemente in diesen *struct*s wird wie folgt zugegriffen:

```
quad1.obenlinks.x = 10;
```


Strukturen und Arrays

Strukturen können Arrays enthalten

```
struct beispiel {
 float x[10];
 int y[5];
} bsp1;
bsp1.y[4] = 14;  // Zugriff auf das 5. y-Element
bsp1.x[0] = 3.3F; // Zugriff auf das erste x-Element
```

Arrays von Strukturen

```
struct eintrag {
 char vorname[20];
 char nachname[20];
};
struct eintrag liste[3];
printf("%s\n", liste[1].vorname);
```

Unions

... sind ähnlich wie Strukturen. Es kann aber nur ein Element zu einem bestimmten Zeitpunkt verwendet werden, da alle Elemente den selben Speicherbereich nutzen.

```
union teil {
 char a;
 int z;
} teil1;
teil1.z = 10;
```


typedef

Mit **typedef** kann man ein Synonym für den Namen eines Datentyps, z.B. eines structs festlegen. Über diesen Namen kann dann auf den Datentyp zugegriffen werden.

```
typedef struct {
 int x;
 int y;
} koord;

koord obenrechts;
```

In der Wirkung entspricht **typedef** einem Präprozessor-Makro (mit vertauschter Ersetzung), ist aber tatsächlich Bestandteil der Sprache C.

enum (Aufzählung)

```
enum Wochentag
 {
  Montag, Dienstag, Mittwoch, Donnerstag,
  Freitag, Samstag, Sonntag
 };
...

enum Wochentag w = Montag;

if(w == Montag) printf("Heute ist Montag\n");
```


Call by value

```
#include <stdio.h>

void SetzeAuf10(int a) { a = 10; }

int main()
{
 int zahl=1;

 SetzeAuf10(zahl);
 printf("%d\n",zahl);
 return 0;
}
```

Frage: Was gibt dieses Programm aus?

Zeigervariablen

Parameter in Funktionen werden in C (mit Ausnahme von Feldvariablen, warum ☞ sehen wir später) immer als **Wert** übergeben. Die Parameter-Variablen selbst sind stets nur *lokale* Variablen in der Funktion. Wenn sie geändert werden, ändert sich der Wert einer Variablen im aufrufenden Programm nicht!

Eine Zeigervariable hingegen ist eine Angabe der Speicheradresse, an der ein Wert liegen kann. Wird etwas in diese Speicheradresse geschrieben, so ändert sich der dort gespeicherte Wert.

```
void SetzeAuf10(int *a) { *a = 10;}
```


Zeiger und Feldnamen

Die **Namen** von Datenfeldern (Arrays) ohne die `[ ]` sind implizit **Zeiger** auf das erste Element des Datenfeldes.

```
int feld[10];
```

`feld` im Programmtext bezeichnet das gleiche wie `&feld[0]`, nämlich die **Adresse** des ersten Elementes. Daher können die Inhalte von Datenfeldern der aufrufenden Funktion von der aufgerufenen Funktion verändert werden, da automatisch über Zeiger auf die Inhalte zugegriffen wird. **&** ist hier der **Adressoperator**, der die Adresse der nachfolgenden Variablen zurückgibt.

Index-Zeigernotation bei Arrays

Statt der Indexnotation von Arrays können die einzelnen Elemente der Arrays über Zeiger angesprochen werden. Dabei ist *Arrayname* ohne eckige Klammern ein Zeiger auf das erste Element.
Die einzelnen Elemente eines Arrays entpsicht folgender Zeiger-Notation:

```
*(array)     == array[0]
*(array + 1) == array[1]
*(array + 2) == array[2]
*(array + 3) == array[3]
*(array + n) == array[n]
```


Zeiger auf Strukturen

```
struct ballon {
 char farbe[10];
 float gewicht;
};
struct ballon *z_ballon; // Zeiger auf Struktur
struct ballon meinballon; // Instanz der Struktur
z_ballon = &meinballon; // Zeiger-Initialiserung
(*z_ballon).gewicht = 300F; // Zuweisung über Zeiger
z_ballon->gewicht = 300F; // Häufigere Schreibweise
```

Arrays als Funktionsparameter

Um innerhalb einer C-Funktion mit Arrays arbeiten zu können, sollte der Funktion nicht nur ein Zeiger auf das erste Element des Arrays, sondern auch die Anzahl der Elemente im Array übergeben werden, wenn sich diese nicht aus bestimmten Werten im Array ermitteln lässt (z.B. *0* als letztes Element in Zeichenketten-Arrays).

```
int maximum(int *x, int anzahl)
{
 int i, max = x[0];
 for (i = 0; i < anzahl; i++)
  {
    if (x[i] > max) max = x[i];
  }
 return max;
}
```


Zeigerarithmetik

Werden Zeiger inkrementiert, so zeigen sie auf das **nächste Element** eines Datenfeldes, bzw. auf die nächste verfügbare Speicheradresse, in die ein solches Element abgelegt werden kann.
Beispiel: Kopieren einer Zeichenkette (String)

```
// Achtung: Strings sind immer durch ein 0-Zeichen
// terminiert!
char a[14] = "Hello, World!";
char b[14];
char *p1 = &a[0]; // Zeiger auf erstes Element von a[].
char *p2 = b;     // macht das gleiche.

// Kopiert jedes Zeichen von a nach b, bis zum 0-Zeichen
while(*p2++ = *p1++);
```

Call by Reference

```
void SetzeAuf10(int& a) { a = 10;}
```

In C++ (!) ist auch eine Übergabe von Variablenreferenzen als Funktionsparameter möglich, d.h. es wird nicht der **Wert**, sondern die **Variable selbst** übergeben, und kann auch innerhalb der Funktion verändert werden. In C gibt es diese Möglichkeit nicht, daher werden in C oft Zeiger verwendet, um auf Variablen gemeinsam von mehreren Funktionen aus zuzugreifen.

Die globale Deklaration von gemeinsamen Variablen (außerhalb von **`main()`** oder jeder anderen Funktion), ist zwar möglich, aber gilt als schlechter Programmierstil.

Zeiger auf Zeiger

Bei einem Zeiger auf Zeiger wird bei der Deklaration und der Verwendung des Zeigers der Indirektionsoperator ***** doppelt genutzt. Für jede Indirektion kommt ein weiterer ***** dazu.

```
int *zgr, x; // Zeiger zgr und int-Variable x
zgr = &x; // Zeiger zeigt auf x
*zgr = 12 // x wird 12 zugewiesen
int **zeiger_auf_zgr  = &zgr;
**zeiger_auf_zgr = 12 // entspricht *zgr = 12

int d2array[2][4]; // zweidimensionales Array
/* d2array zeigt auf ein Array d2array[0]
   d2array[0] zeigt auf d2array[0][0] */
```

Arrays von Zeigern und Strings

Ein Array ist, wie wir gesehen haben, ein zusammenhängender Block von Speicherstellen. Oft werden Arrays von Zeigern bei sog. Strings (=Zeichenketten) eingesetzt. Der Beginn eines Strings ist ein Zeiger auf das erste Zeichen (char). Mehrere Strings können mit Arrays von Zeigern auf char verwaltet werden, wobei jedes Array-Element auf einen String zeigt.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
 int i
 for(i=0; i< argc; i++)
   printf("Argument %d: %s, Länge=%d\n",
          i, argv[i], strlen(argv[i]) );
 return 0;
}
```


Zeiger auf Funktionen

Bei der Programmausführung eines Programmm wird der Code der Funktion in den Speicher geladen. Jede Funktion erhält dann eine Startadresse, wo deren Code beginnt. Ein Zeiger auf eine Funktion ist dann ein Zeiger auf diese Startadresse.

```
/* Zeiger auf funk1 mit int-Parameter, Rückgabetyp int */
int (*funk1)(int x);

/* Zeiger auf funk2 mit 2 double-Param. ohne Rückgabewert */
void (*funk2)(double y,double z);

int rechteck(int x); /* Funktionsprototyp */
/* Damit man einen Zeiger auf eine Funktion setzen kann,
   müssen die Parameterliste und der Rückgabewert
   übereinstimmen */
funk1 = rechteck;  // Initialisieren des Zeigers
ergebnis = funk1(x) // Aufruf der Funktion über den Zeiger
```

Verkettete Listen

```
struct element
{
  char inhalt[100];
  struct element *nachfolger;
}
```

Mit verketten Listen lassen sich Elemente unbekannter Zahl “aneinanderhängen“, was bei Feldern aufgrund einer vorher zu kennenden Maximalanzahl äußerst schwierig wäre. Jedes Element enthält mindestens einen Zeiger auf das nachfolgende Element (Nachfolger).

Gültigkeitsbereiche

Als Gültigkeitsbereich bezeichnet man den Bereich im Quellcode, in dem eine Variable „sichtbar“, also verwendbar ist. Zudem wird die Lebensdauer von Variablen festgelegt, also wie lange sie im Speicher vorhanden sind.

Man unterscheidet zwischen globalen Variablen, die überall im Programm sichtbar sind und lokalen Variablen, die nur innerhalb einer Funktion oder eines Programmblocks definiert ist.

Globale Variablen

```
#include <stdio.h>
int a = 666; // a ist global und überall sichtbar

void wert_anzeigen(void)
{
 printf("%d\n",a);
}

int main()
{
 printf("%d\n",a);
 wert_anzeigen();
 return 0;
}
```


lokale Variablen

```
#include <stdio.h>

int main()
{
 int zahl = 0; // Variable lokal zu main()
 int i;
 printf("außerhalb des Blocks, zahl = %d\n",zahl);
 for (i=0;i<10;i++)
 {
   int zahl = 20;
   printf("innerhalb des Blocks, zahl = %d\n",zahl);
 }
 return 0;
}
```

statische lokale Variablen

... behalten zwischen den Funktionsaufrufen ihren Wert.

```
#include <stdio.h>

void hochzaehlen(void) {
 static int x = 0; static int y = 0;
 printf("x = %d, y = %d\n", x++, y++);
}

int main() {
 int count;
 for(count=0; count<10; count++)
  {
   printf("Im Durchlauf %d: ", count); hochzaehlen();
  }
 return 0;
}
```


weitere Speicherklassen

statische globale Variablen: Im Gegensatz zu normalen globalen Variablen sind diese nur in Funktionen in der eigenen Datei sichtbar und nicht von Funktionen aus anderen Dateien nutzbar
Variablen mit dem Schlüsselwort ’extern’ sind statische gloabe Variablen, die an einer anderen Stelle im Programm definiert werden.

Registervariablen: lokale Variablen mit dem Schlüsselwort ’register’ werden nicht im Arbeitsspeicher, sondern in einem Prozessorregister, sofern möglich, abgelegt.

cast: Typumwandlungen

Besitzen zwei Operanden in einem Ausdruck verschiedene Datentypen, so ist das Ergebnis vom Typ mit dem größeren Wertebereich. Man unterscheidet bei der Typumwndlung zwischen impliziten und expliziten casts.

```
int x = 3; int a = 2;
float y = 6.3F;
float z;

z = y / x; // x wird implizit in float umgewandelt
z = x; // x erhält für die Zuweisung den Typ float
x = y; // y wird implizit in int umgewandelt, x = 6
z = x/a; // z = 1, da Ganzzahldivision
z = (float)x/a // expliziter cast von x, z = 1.500000
```


Speicherreservierung

Mit der Funktion malloc() kann man Speicher reservieren. Als Argument übergibt man die Größe in Bytes. Als Rückgabewert wird eine Adresse des reservierten Speicherblocks zurückgeben, die vom Typ **`void *`** (=„Zeiger unbestimmten Typs“) ist.

```
/* Speicher holen für ein Array mit 5 float-Werten */
float *zahlen;
zahlen = (float *) malloc(5 * sizeof(float));

/* Speicher holen für ein Array mit 50 int-Werten */
int *zahlen2;
zahlen2 = (int *) malloc(50 * sizeof(int));

/* Speicher wieder freigeben */
free(zahlen); free(zahlen2);
```

Die shift-Operatoren

...verschieben die Bits von Integer-Variablen um eine bestimmte Anzahl an Positionen im Dualsystem. **x << n** verschiebt die Bits der Variablen **x** um **n** Stellen nach links, **x >> n** um n Stellen nach rechts.

```
int x = 12; // binär 00001100
// Verschieben um 1 nach rechts,
// entspricht dem Teilen durch 2
x = x >> 1; // binär 00000110,
printf("x = %d\n",x); // x = 6

int y = 48; // binär 00110000
// Verschieben um 2 nach links,
// entspricht dem Multiplizieren mit 4
y = y << 2; // 11000000
printf("y = %d\n",y); // y = 192
```


logische Bitoperatoren

Dazu gehören **&** (bitweises AND), **|** (bitweises OR), ^ (bitweises XOR) und der Komplement-Operator ~.

```
int x = 240; // binär 11110000
int y = 85;  // binär 01010101

int a = x&y; // binär 01010000
int b = x|y; // binär 11110101
int c = x^y; // binär 10100101
int d = ~x;  // binär 00001111

printf("a = %d\n",a); // a = 80
printf("b = %d\n",b); // b = 245
printf("c = %d\n",c); // c = 165
printf("d = %d\n",d); // d = 15
```

Formatierte Ausgabe

```
#include <stdio.h>
int main() {
 float zahl = 12334.222;
 int n;
 /* Gibt Zahl mit Feldlänge 5 aus */
 printf("%5f\n",zahl);
 /* Mit dem Feldlängenspezifizierer * wird die
 Feldlänge aus der Argumentliste übernommen */
 for (n=5;n<=20;n+=5)
   printf("%*f\n",n,zahl)
 /* - richtet Werte linksbündig aus */
 /* # stellt oktalen bzw. hexadezimalen
    Zahlen 0 bzw. 0X voran */
 printf("%-15d%-#15o%-#15X\n", zahl,zahl,zahl);
 return 0;
}
```


Streams

Die gesamte Ein- und Ausgabe erfolgt bei C über sogenannte Streams. Ein Stream ist eine Folge von Bytes, die in ein oder aus einem Programm hinein- bzw. herausfließen. Die Nutzung der Streams ist geräteunabhängig.
Achtung: Viele der auf der Standardeingabe **`stdin`** arbeitenden Zeichen-Funktionen liefern statt **`char`** ein **`int`** zurück!

```
#include <stdio.h>
int main() {
 int ch;
 while ((ch = getchar()) != '°') putchar(ch);
 return 0;
}
```

Dateiverwaltung

Die Dateiverwaltung erfolgt in C über Streams. Dabei gibt 2 Arten von Dateien, Textdateien und binäre Dateien. Die Textdateien sind dabei zeilenorientiert. Alle Dateinamen werden in Strings gespeichert, welche Zeichen für Namen hier erlaubt sind, ist vom Betriebsystem abhängig. Dateinamen sollten bei der Deklaration mit dem gesamten Pfad angegeben werden.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
char *dateiname = "/testdateien/test.txt";
FILE *fp /* Zeiger auf eine Struktur, welche für die
            Steuerung von Dateioperationen notwendig ist */
if (fp = fopen(dateiname,"rw")) != NULL
  {
    printf ("%s erfolgreich geöffnet",dateiname);
    fclose(fp); // Schließen der Datei
  }
else
    fprintf(stderr, "Fehler beim Öffnen von %s\n", dateiname);
}
```


Dateien lesen und schreiben

Es gibt 3 Arten zum Lesen und Schreiben von Dateien, formatiert mit **`fprintf`** und **`fscanf`**, zeichenorientiert mit **`gets`** bzw. **`fgets`** und **`puts`** bzw. **`fputs`** und direkt mit **`fread`** und **`fwrite`**.

```
// Beispiel für direktes Lesen und Schreiben von Dateien
#include <stdio.h>
#include <stdlib.h>
#define GROESSE 30
int main() {
 int count, array[GROESSE], array2[GROESSE];
 FILE *fp;
 for (count = 0; count<GROESSE; count++) array1[count] = 2*count;
 if ( (fp=fopen("direkt.txt","wb")) == NULL)
    {fprintf(stderr, "Fehler beim Öffnen der Datei\n"); exit(1); }
 if ( (fwrite(array1,sizeof(int),GROESSE,fp)) != GROESSE)
    {fprintf(stderr, "Fehler beim Schreiben in die Datei\n"); exit(1); }
 if ( (fread(array2,sizeof(int),GROESSE,fp)) != GROESSE)
    {fprintf(stderr, "Fehler beim Lesen der Datei\n"); exit(1); }
 fclose(fp);
 return(0);
}
```

Weitere Dateioperationen

eine Datei löschen mit remove

```
char *datei = "wegdamit.txt";
if (remove(wegdamit) == 0)
  printf("Die Datei %s wurde gelöscht.",datei);;
```

Eine Datei umbennen mit rename

```
char *altername = "alt.txt", *neuername = "neu.txt";
if (rename(altername,neuername) == 0)
  printf("%s wurde in %s umbenannt",
          altername, neuername);
```