Programmieren

C unter UNIX

Witersemester

Thoerie-Unterricht

1. „Hallo Welt“ in C

Was benötige ich für C?

- ASCII Texteditor (vi oder Texteditor im KDE)
- Compiler (gcc, cc, …)
- Oder Entwicklungsumgebung (Compiler & Texteditor in einem)

```c
/* Hallo-Welt.c */

#include <stdio.h>

int main (void) {
      printf("Hallo Welt\n");
      getchar();
      return 0;
}
```

Zunächst wird das Programm in einem Verzeichnis, wo immer das Programm zur Ausführung gebracht werden soll, mit der Endung - auch Extension genannt - "*.c" abgespeichert. Jetzt müssen Sie das Programm übersetzen (kompilieren).

(unter Windows immer getchar(); vor return 0)

Kommandozeile in Linux: `gcc -o BSP BSP.c`

2. Analyse

include ist kein direkter Bestandteil der Sprache C, sondern ein Befehl des Präprozessors. Der Präprozessor ist ein Teil des Compilers, der nicht das Programm übersetzt, sondern kontrollierend temporäre Änderungen im Programmtext vornimmt. Es ist in jedem Programm, das mit der Konsole arbeitet, nötig.

main ist die Hauptmethode von welcher das Programm gestartet wird.

printf gibt etwas in die Konsole aus, der Befehl \n sorgt weiters dafür, das bei der nächsten Ausgabe ein Leerzeichen dazwischen ist.

getchar sorgt für eine Unterbrechung des Programms bis „Enter“ gedrückt wird.

3. Zeichensätze

3.1. BASIC-Zeichensätze

- Die zehn Dezimalziffern 0-9
- Die Buchstaben des englischen Alphabets: A – Z und a - z
- Die folgenden Grafiksymbole:
  ! " % & / ( ) [ ] { } \ ? =
  ' # + * ~ - _ . : ; , | < > ^
- Die Whitespace-Zeichen: Leerzeichen, Tabulatorzeichen, neue Zeile, neue Seite

Auf manchen PCs (nicht ANSI C) kann auch das Dollarzeichen ($) verwendet werden.

3.2. Steuerzeichen

Steuerzeichen beginnen immer mit einem Backslash (\) und einer folgenden Konstante. Der Compiler behandelt diese Steuerzeichen wie ein einziges Zeichen.

Steuerzeichen	**Bedeutung**
\a BEL (bell)	akkustisches Warnsignal
\b BS (backspace)	setzt Cursor um eine Position nach links
\f FF(formfeed)	ein Seitenvorschub wird ausgelöst. Wird hauptsächlich bei Programmen verwendet, mit denen Sie etwas ausdrucken können.
\n NL (newline)	Cursor geht zum Anfang der nächsten Zeile
\r CR (carriage return)	Cursor springt zum Anfang der aktuellen Zeile
\t HT (horizontal tab)	Zeilenvorschub zur nächsten horizontalen Tabulatorposition (meistens acht Leerzeichen weiter)
\v VT (vertical tab)	Cursor springt zur nächsten vertikalen Tabulatorposition
\"	" wird ausgegeben
\’	' wird ausgegeben
\?	? wird ausgegeben
\\	\ wird ausgegeben
\0	ist die Endmarkierung eines Strings
\nnn	Ausgabe eines Oktalwerts (z.B. \033 = ESCAPE-Zeichen)
\xhh	Ausgabe eines Hexdezimalwerts

4. Kommentare

Sie werden einfach beginnend mit zwei // oder zwischen /* … */ geschrieben. In den Kommentaren können Sie beliebige Zeichen verwenden, also auch deutsche Umlaute oder das Dollarzeichen. Es muss jedoch beachtet werden, das der Kommentar nicht mir \ endet.

5. Formatierte Eingabe mit scanf()

Mit der Funktion scanf() können Werte unterschiedlicher Datentypen formatiert Eingelesen werden. Eingelesen wird dabei von der Standardeingabe (stdin). Mit Standardeingabe ist normalerweise die Tastatur gemeint.

```
/* scanf1.c */
#include <stdio.h>

int main (void) {
        int i;                  /* Ein ganzzahliger Datentyp */
        printf("Bitte geben Sie eine Zahl ein : ");
        scanf("%d",&i);         /* Wartet auf die Eingabe */
        printf("Die Zahl, die Sie eingegeben haben, war %d\n",i);
return 0;
}
```

Wenn das Programm korrekt abläuft, wird nach einer Zahl gefragt. Jetzt gibt der Anwender eine Zahl ein und drückt ENTER. Anschließend gibt das Programm die Zahl, die eingegeben wurde, auf dem Bildschirm aus und wird beendet.

Ein häufiges Problem, das auftritt, wenn Sie scanf() für die Eingabe verwenden, ist die Pufferung. Diese ist je nach System und Anwendung zeilen- oder vollgepuffert. Dies gilt wiederum nicht für die Standardfehlerausgabe (stderr), die laut ANSI C niemals vollgepuffert sein darf.

```
/* scanf2.c */
#include <stdio.h>

int main(void) {
   char a,b,c;
   printf("1. Buchstabe : ");
   scanf("%c",&a);
   printf("2. Buchstabe : ");
   scanf("%c",&b);
   printf("3. Buchstabe : ");
   scanf("%c",&c);
   printf("Sie gaben ein : %c %c %c ",a,b,c);
   return 0;
}
```

Lösung: `fflush(stdin)`

Sie benutzen die Funktion fflush() zum Entleeren des Tastaturpuffers. Möglicherweise gelingt dies nicht auf jedem Betriebssystem (speziell nicht unter Linux):

```
/* scanf3.c */
#include <stdio.h>

int main(void) {
   char a,b,c;
   printf("1. Buchstabe : ");
   scanf("%c",&a);
   fflush(stdin);
   printf("2. Buchstabe : ");
   scanf("%c",&b);
   fflush(stdin);
   printf("3. Buchstabe : ");
   scanf("%c",&c);
   printf("Sie gaben ein : %c %c %c ",a,b,c);
   return 0;
}
```

6. Der Adressoperator "&"

Den Adressoperator "&" jetzt schon besser zu verstehen, kann nicht schaden. Später, wenn das Thema der Pointer besprochen wird, kann dieses Vorverständnis von Nutzen sein.

```
/* FALSCH, da Adressoperator & fehlt */
scanf("%d", zahl);

/* Richtig, eine Zeichenkette benötigt keinen Adressoperator */
scanf("%s", string);
```

Auch wenn scanf() das Gegenstück zu printf() ist und sich beide in ihrer Schreibweise ähneln, sollten Sie nicht auf die Idee kommen, Folgendes zu schreiben:

```
/* FALSCH */
scanf("Bitte geben Sie eine Zahl ein: %d\n", &zahl);
```

Das funktioniert deshalb nicht, weil scanf() für die Standardeingabe programmiert ist und printf() für die Standardausgabe. Wobei die Standardausgabe auf der Kommandozeile auch umgeleitet werden kann. Aber dazu später mehr.

7. Formatierte Ausgabe mit printf()

Der Rückgabewert von printf() ist die Anzahl der Zeichen, die ausgegeben werden. printf() bekommt mindestens einen Parameter, nämlich den Formatstring. Bei dem Formatstring handelt es sich um einer Zeichenkette beliebiger Länge.

Die Funktion printf() wird bei der Ausführung von rechts nach links abgearbeitet. Dabei sucht die Funktion nach einem Ausgabetext (Stringkonstante) und Formatanweisungen. Formatanweisungen beginnen alle mit einem %-Zeichen. Dahinter folgt ein Buchstabe, der den Datentyp des Formates angibt. %d steht z.B. für eine dezimale Ganzzahl.

```
/* printf1.c */
#include <stdio.h>

int main(void) {
   int zeichen;
   zeichen=printf("Hallo Welt");
   printf(" enthaelt %d Zeichen\n",zeichen);  //10 Zeichen
   return 0;
}
```

8. Elementare Datentypen

8.1. Variablen deklarieren

Eine Variable wird in C genau wie in C# deklariert, nämlich wie folgt:

Typ Variablenname;

Hier eine Liste der möglichen Bezeichner

- Ein Bezeichner darf aus einer Folge von Buchstaben, Dezimalziffern und Unterstrichen bestehen. Einige Beispiele:
  var8, _var, _666, var_fuer_100tmp, VAR, Var
- C unterscheidet zwischen Groß- und Kleinbuchstaben.
  Var, VAr, VAR, vAR, vaR, var
  Hierbei handelt es sich jeweils um verschiedene Bezeichner.
- Das erste Zeichen darf keine Dezimalzahl sein.
- Die Länge des Bezeichners ist beliebig lang. Nach ANSI C-Standard sind aber nur die ersten 31 Zeichen bedeutend. Obgleich viele Compiler auch zwischen mehr Zeichen unterscheiden können.
- Variablen müssen immer ganz oben deklariert werden (ANSI99-Standard, da es weiter unten in C nicht mehr möglich ist (im Gegensatz zu C++/C#)

8.2. Der Datentyp int (Integer)

Die Größe des Datentypes int ist vom Prozessor abhängig.
Will man feststellen wie groß int beim eigenen System ist, kann man mit sizeof() die Größe abfragen.

```
int a;  // Definition
```

Name	Größe	Wertebereich	Formatzeichen
int	4 Byte	-32768 +32767	%d oder %i

8.3. Der Datentyp long

Long ist größer als ein Integer, verbraucht aber auch mehr Speicherplatz.
Will man den Datentyp int vergrößern so schreibt man vor den Typ ein long:

long int variablenname

Name	Größe	Wertebereich	Formatzeichen
long	4 Byte	-2147483648 +2147483647	%ld oder %li

8.4. Der Datentyp short

Short ist kleiner als int und long aber auch weniger Speicherintensiv.

Name	Größe	Wertebereich	Formatzeichen
short	2 Byte	-32768 +32767	%d oder %i

8.5. Die Gleitpunkttypen float und double

Die Datentypen float und double werden verwendet wenn man Gleitkommazahlen(eigentlich ja Gleitpunkttypen) darstellen will(sie können übrigens auch mit long erweitert werden).

Name	Größe	Wertebereich	Genauigkeit	Formatzeichen
float	4 Byte	1.2E-38 3.4E+38	6-stellig	%f
double	8 Byte	2.3E-308 1.7E+308	15-stellig	%lf
long double	10 Byte	3.4E-4932 1.1E+4932	19-stellig	%Lf

8.6. Der Datentyp char

Der Datentyp char wird primär verwendet um Buchstaben zu speichern, kann aber auch kleine Zahlen speichern(Buchstaben können mit ASCII-Code oder unter einfachem Hochkomma zugewiesen werden).

Name	Größe	Wertebereich	Formatzeichen
char	1 Byte	-128 +127	%d oder %i

8.7. Vorzeichenlos und vorzeichenbehaftet

Durch das Schlüsselwort „unsigned“ werden Variablen als nicht vorzeichenbehaftet eingestuft was den positiven Wertebereich von Ganzzahlvariablen vergrößert. Das Gegenteil „signed“ ist voreingestellt.

Name	Größe	Wertebereich	Formatzeichen
unsigned char	1 Byte	0 bis 255	%d oder %i oder %x oder %u

8.8. Konstanten

Eine Konstante ist entweder eine ganzzahlige Konstante, eine Gleitpunktkonstante, Zeichenkonstante oder ein String-Literal. Jede Konstante besteht aus einem Typ, der sich aus dem Wert und seiner Schreibweise ergibt.

8.9. Umwandlungsvorgaben für formatierte Ein-/Ausgabe

Zwar wurden schon ein paar Formatzeichen bei scanf/printf verwendet, doch gibt es noch einige mehr. Die genaue Liste ist auf pronix.de nachzulesen. Man kann jedoch sagen dass diese Formatzeichen wie sonst auch nach dem „%“ stehen.

Name	Formatzeichen
int	%d oder %i
short	%d oder %i
long	%ld oder %li
float	%f
double	%lf
long double	%Lf
char	%c
char[]	%s
	%2.2x

9. Operatoren

9.1. Exkurs zu Operatoren

Operatoren können :

- unär - der Operator hat einen Operanden.
- binär - der Operator hat zwei Operanden.
- tertiär - der Operator hat drei Operanden.

sein ,aber auch ein :

- Infix - der Operator steht zwischen den Operanden.
- Präfix - der Operator steht vor den Operanden.
- Postfix - der Operator steht hinter den Operanden.

sein.

Links-/Rechtsassoziativ bezieht sich bei Operatoren auf die Richtung nach der die Berechnungen durchgeführt werden.

9.2. Arithmetische Operatoren

Genauso wie in C#

Operator	Bedeutung
+	addiert zwei Werte
-	subtrahiert zwei Werte
*	multipliziert zwei Werte
/	dividiert zwei Werte
%	Modulo (Rest einer Division)

9.3. Erweiterte Darstellung arithmetischer Operatoren

Dies ist ebenso wie in C#. Falls vor dem = ein Operator(+ , - ,* , /) steht so wird der Wert rechts vom = dem Wert links vom = zugeteilt.

`x = x + 3;` Kurzschreibweise: x += 3;

9.4. Inkrement- und Dekrement-Operatoren

Oh Wunder ebenso wie in C#. Vor der zu dekrementierenden/inkrementierenden Variable muss man zwei +/- schreiben.

Operator	Bedeutung
++	Inkrement
--	Dekrement
var++	Postfix-Schreibweise
++var	Präfix-Schreibweise
var--	Postfix-Schreibweise
--var	Präfix-Schreibweise

a=5;

b=a++; von rechts nach links abgearbeitet b=a; a++;

c=++a; von rechts nach links abgearbeitet a++; c=a;

Schneller als a = a +1

9.5. Vergleichsoperatoren

==	ist gleich
!=	nicht ist; ungleich
<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich

9.6. Logische Operatoren

&&	und	if ((Bedingung1) && (Bedingung2))
||	oder	if((Bedingung1) || (Bedingung2))
!	nicht	if(! (Bedingung1))

9.7. Bit-Operatoren

Bitoperator	Formatzeichen
&, &=	Bitweise AND-Verknüpfung
|, |=	Bitweise OR-Verknüpfung
^, ^=	Bitweise XOR
~	Bitweises Komplement
>>, >>=	Rechtsverschiebung
<<, <<=	Linksverschiebung

9.7.1. Bitweises UND &

Steht der &-Operator zwischen zwei Operanden, so handelt es sich um den bitweisen UND-Operator. Dieser ist leicht mit dem unären Adressoperator zu verwechseln.

Der Operator wird hauptsächlich dafür verwendet, einzelne Bits gezielt zu löschen.

```
char x= 3;
x= x  &  1;          // alles außer 1.Bit wird gelöscht!
```

9.7.2. Bitweises ODER |

Mit dem bitweisen ODER-Operator können Sie gezielt zusätzliche Bits setzen. Verwendet wird dieser wie schon zuvor der bitweise UND-Operator:

```
char x = 1;
x = x|126;     // x=127
```

9.7.3. Bitweises XOR ^

Dieser exklusive ODER-Operator liefert nur dann eine 1 zurück, wenn beide Bits unterschiedlich sind. Er ist sehr gut geeignet, um Bits umzuschalten. Alle gesetzten Bits werden gelöscht und alle gelöschten gesetzt.

```
char x=20;
x = x^55;     // x=35
```

9.7.4. Bitweises Komplement ~

Der NOT-Operator (~) wirkt sich auf Zahlen so aus, dass er jedes einzelne Bit invertiert.

```
char x=20;
x=~x;  /* x= -21 */
```

9.7.5. Linksverschiebung <<

Mit einer Linksverschiebung (<<) werden alle Bits einer Zahl um n Stellen nach links gerückt. Die rechts entstehenden Leerstellen werden mit 0 aufgefüllt (Ergebnis bei – Wert compilerspezifisch). Bei einer Stelle wird ein * mit 2, bei zwei mit 4, usw. durchgeführt.

9.7.6. Rechtsverschiebung >>

Gegenstück zur Linksverschiebung. Hier wird durch 2^x dividiert.

9.7.7. Beispiele

4. Bit auf “1” setzen

```
reg = reg | 0x08; bzw. reg = reg | 8;
```

3. und 5. Bit auf “0” setzen

```
reg = reg & 0xEB;
reg = reg & ~0x
```

6. Bit umschalten

```
reg = reg ^ 0x20;
```

7. Bit abfragen, ob auf "1"

```
if(reg&64)
```

Funktion – Parameter: Bitposition Register; Das angeg. Bit auf „1" setzen

```
tmpreg = 0x01;
tmpred = tmpreg << pos;
reg = reg | tmpreg;
```

9.7.8. Rezept für Fortgeschrittene

```
int wert;
unsigned char reg;
scanf(„%d",&wert);
reg=(char)wert;
```

Register als Bit-Folge ausgeben:

```
int hilf,i;
hilf=reg;
for (i=0; i<8; i++)
   {
    if (hilf & 128) printf("1");
       else printf("0");
   hilf=hilf <<1;
  }
printf("\n");
```

```
int hilf,i;
hilf=reg;
for (i=0; i<8; i++)
   {
     printf("%d",hilf & 128);
     hilf=hilf <<1;
  }
printf("\n");
```

Die Anzahl der gesetzten Bits ausgeben:

```
int hilf,i;
int summe=0;
hilf=reg;
for (i=0; i<8; i++)
   {
    if (hilf & 128) summe++;
   hilf=hilf <<1;
  }
printf("Summe: %d\n",summe);
```

Oft ist eine Funktion wünschenswert, mit der eine Zahl daraufhin getestet wird, ob ein bestimmtes Bit gesetzt ist, oder mit der sich gezielt einzelne Bits setzen oder löschen lassen. Hierzu ein Listing mit entsprechenden Funktionen:

```
/* playing_bits.c */
#include <stdio.h>
#define BYTE unsigned char
/* Funktion : Bit_Test()
* val : der Wert, den es zu testen gilt
* bit : Bitnummer, die abgefragt wird, ob gesetzt (0-7)
* Rückgabewert : (1)=Bit gesetzt; (0)=Bit nicht gesetzt
*/
int Bit_Test(BYTE val, BYTE bit) {
BYTE test_val = 0x01; /* dezimal 1 / binär 0000 0001 */
/* Bit an entsprechende Pos. schieben */
test_val = (test_val << bit);
/* 0=Bit nicht gesetzt; 1=Bit gesetzt */
if ((val & test_val) == 0)
return 0; /* Nicht gesetzt */
else
return 1; /* gesetzt */
}
/* Funktion : Bit_Set()
* val : Wert, bei dem Bit gesetzt werden soll
* bit : Bitnummer, die gesetzt werden soll (0-7)
* Rückgabewert : keiner
*/
void Bit_Set(BYTE *val, BYTE bit) {
BYTE test_val = 0x01; /* dezimal 1 / binär 0000 0001 */
/* Bit an entsprechende Pos. schieben */
test_val = (test_val << bit);
*val = (*val | test_val); /* Bit an Pos bit setzen */
}
/* Funktion : Bit_Clear()
* val : Wert, bei dem Bit gelöscht werden soll
* bit : Bitnummer, die gelöscht werden soll (0-7)
* Rückgabewert : keiner
```

```
*/
void Bit_Clear(BYTE *val, BYTE bit) {
BYTE test_val = 0x01; /* dezimal 1 / binär 0000 0001 */
/* Bit an entsprechende Pos. schieben */
test_val = (test_val << bit);
*val = (*val & (~test_val)); /* Bit an Pos bit löschen*/
}
int main(void) {
BYTE wert = 0;
/* Test, ob Bit 0 gesetzt */
printf("%s\n",Bit_Test(wert, 0)?"gesetzt": "nicht gesetzt");
Bit_Set(&wert, 0); /* Bit 0 setzen */
/* Wieder testen, ob Bit 0 gesetzt */
printf("%s\n",Bit_Test(wert, 0)?"gesetzt": "nicht gesetzt");
Bit_Clear(&wert, 0); /* Bit 0 wieder löschen */
/* Wieder testen ob Bit 0 gesetzt */
printf("%s\n",Bit_Test(wert, 0)?"gesetzt": "nicht gesetzt");
return 0;
}
```

9.8. *sizeof-Operator*

Der sizeof – Operator liefert den Größenbereich eines Datentypes in Byte zurück(simpel aber gut :D).

```c
#include <stdio.h>

int main(void) {
   printf("char     : %d Byte\n" , sizeof(int));
   printf("Hallo    : %d Bytes\n", sizeof("Hallo"));
   printf("A        : %d Bytes\n", sizeof((char)'A'));
   printf("34343434 : %d Bytes\n", sizeof(34343434));
   return 0;
}
```

10. Typenumwandlung

Einen Datentyp in C zu ändern kann man auf zwei Arten erzielen.

1.) Impliziter Typecast – wird vom Compiler selbst durchgeführt.
2.) Expliziter Typecast – muss man folgend durchführen:
    (typ) ausdruck

```c
#include <stdio.h>

int main(void) {
   int x = 5, y = 2;
   float z;

   z = x / y;
   printf("%f\n", z);          /* = 2.000000 */
   z = (float) x / (float) y;    /* = 2.500000 */
   printf("%f\n", z);
   return 0;
}
```

11. Kontrollstrukturen

11.1. Verzweigungen mit der if-Bedingung

```
Ist genau wie in C#.

if(Ausdruck){Anweisungsblock}
[else{Anweisungsblock}]
```

11.2. Bedingungsoperator :

kann man als Kurzschreibweise für if-else hernehmen.

Syntax: `<BEDINGUNG> ? <AUSDRUCK1> : <AUSDRUCK2>`

```
max= a>b ? a : b


z.B.: Tagesarbeitszeit:
MO bis DO => 8 h
FR        => 6 h
SA und SO => 0 h


std= Tag>5; 0; Tag == 5; 6 : 8;
```

11.3. Fallunterscheidung: die switch-Verzweigung

Funktioniert wie in C# , wobei beim default-Zweig kein break gebraucht wird.

```
switch(a) {
      case 1: printf("Das war eins \n");
              break;
      case 2: printf("Das war zwei \n");
              break;
      case 3: printf("Das war drei \n"); // Dieser Fall ist korrekt
      }
```

11.4. Die for-Schleife

```
   int i;
   for(i = 0; <Bedingung> ; i++)
   {
      // Der Code
   }
```

11.5. Die while-Schleife

```
while( <Bedingung> )
   {
      // Der Code
   }
```

11.6. Die do-while-Schleife

```
do {
    // Der Code
   } while( <Bedingung> );
```

11.7. Kontrollierte Sprünge

continue - damit beenden Sie bei Schleifen nur den aktuellen Schleifendurchlauf.

break - beendet die Schleife oder eine Fallunterscheidung. Befindet sich break in mehreren geschachtelten Schleifen, wird nur die innerste verlassen.

exit - beendet das komplette Programm.

return - beendet die Iteration und die Funktion, in der return aufgerufen wird. Im Fall der main()-Funktion würde dies das Ende des Programms bedeuten.

11.8. Direkte Sprünge mit goto

Schlechter Programmierstil → :D

12. Funktionen

12.1. Syntax einer Funktion

```
[Spezifizierer] Rückgabetyp Funktionsname(Parameter) {
         /* Anweisungsblock mit Anweisungen */
}
```

Eine Funktionsdefinition wird in folgende Bestandteile gegliedert:

- Rückgabetyp
- Funktionsname
- Parameter
- Anweisungsblock
- Spezifizierer

12.2. Funktionsdeklaration

Wenn eine Funktion (z.B. hilfe()) erst hinter der main()-Funktion geschrieben wurde, müssen Sie den Compiler zur Übersetzungszeit mit dieser Funktion bekannt machen.

Sonst kann der Compiler in der main()-Funktion mit hilfe() nichts anfangen. Dies stellen Sie mit einer so genannten Vorwärtsdeklaration sicher:

```
/* func1.c */
#include <stdio.h>

void hilfe(void) {
   printf("Ich bin die Hilfsfunktion\n");
}

int main(void) {
   hilfe();
   return 0;
```

12.3. Datenaustausch zwischen Funktionen

```
/* func8.c */
#include <stdio.h>

static int zahl;
void verdoppeln(void);
void halbieren(void);

void verdoppeln(void) {
   zahl *= 2;
   printf("Verdoppelt: %d\n", zahl);
}

void halbieren(void) {
   zahl /= 2;
   printf("Halbiert: %d\n", zahl);
}

int main(void) {
   int wahl;

   printf("Bitte geben Sie eine Zahl ein: ");
   scanf("%d",&zahl);
   printf("Wollen Sie diese Zahl\n");
   printf("\t1.)verdoppeln\n\t2.)halbieren\n\nIhre Wahl: ");
   scanf("%d",&wahl);

   switch(wahl) {
      case 1  : verdoppeln();
                break;
      case 2  : halbieren();
                break;
      default : printf("Unbekannte Eingabe\n");
   }
   return 0;
}
```

12.4. Wertübergabe an Funktionen (call-by-value)

```
/* func9.c */
#include  <stdio.h>
```

```
void verdoppeln(int);
void halbieren(int);

void halbieren(int zahl) {
   zahl /= 2;
   printf("Halbiert : %d\n", zahl);
}

void verdoppeln(int zahl) {
   zahl *= 2;
   printf("Verdoppelt : %d\n", zahl);
}

int main(void) {
   int wahl, z;

   printf("Bitte geben Sie eine Zahl ein : ");
   scanf("%d",&z);
   printf("Wollen Sie diese Zahl\n");
   printf("\t1.)verdoppeln\n\t2.)halbieren\n\nIhre Wahl : ");
   scanf("%d",&wahl);

   switch(wahl) {
      case 1  : verdoppeln(z);
                break;
      case 2  : halbieren(z);
                break;
      default : printf("Unbekannte Eingabe\n");
    }
   return 0;
}
```

12.5. Rückgabewert von Funktionen

```
/* bignum.c */
#include <stdio.h>

int bignum(int a, int b) {
   if(a > b)
      return a;
   else if(a < b)
      return b;
   else
      return 0; /* beide Zahlen sind gleich groß */
}

int main(void) {
   int wert1, wert2, big;
   printf("Bitte einen Wert eingeben: ");
   scanf("%d",&wert1);
   printf("Bitte noch einen Wert eingeben: ");
   scanf("%d",&wert2);
```

```
   big = bignum(wert1, wert2);
   if(big != 0)
      printf("%d ist die größere der beiden Zahlen\n",big);
   else
      printf("Beide Zahlen haben denselben Wert\n");
   return 0;
}
```

12.6. Referenzübergabe an Funktionen (call-by-reference) – Zeiger:

12.6.1. Zeiger (Pointer)

Ein Zeiger wird durch den Stern vor dem Variablennamen gekennzeichnet. Es wird empfohlen, bei der Deklarierung den * immer vor den Pointernamen zu setzen, da es sonst zu Verwirrungen kommen könnte. Der &-Operator liefert die Adresse einer Speicherstelle.

12.6.2. Deklaration

```
int  zahl=3;
int  zahl2=15;

int *zeiger_zahl;
int *zeiger_zahl2;

int *hilfzeiger;
```

12.6.3. Zuweisung

Hierbei ist wichtig, dass es sich bei dem Zugewiesenen um eine Adresse handelt. Dies wird meist durch den Adressoperator (‚&') erreicht.

```
hilfzeiger = &zahl2;
*hilfzeiger=5;
printf(„%d\n", *hilfzeiger);

zeiger_zahl=&zahl;
zeiger_zahl2=&zahl2;

hilfzeiger =zeiger_zahl;   //Zeiger vertauschen
zeiger_zahl=zeiger_zahl2;
zeiger_zahl2=hilfzieger;
```

12.6.4. Zeiger bei Funktionen (Call by reference)

```
int summe(int zahl1, int zahl2, int *erg)
{
        *erg=zahl1 + zahl2;
        return 0;
}
```

```
int zahl1,zahl2,erg2;

Aufruf: summe(zahl1,zahl2,&erg2);
```

12.7. Schlüsselwörter für Variablen – Speicherklassen

12.7.1. auto

Durch voranstellen dieses Schlüsselwortes wird die Variable automatisch angelegt (es ist jedoch überflüssig, da dies sowieso passiert).

12.7.2. extern

Befindet sich die Variable in einer anderen Datei, wird das Schlüsselwort extern davor gesetzt.

12.7.3. register

Dieses Schlüsselwort wird heute überhaupt nicht mehr benutzt.

12.7.4. static

Dieses Schlüsselwort wird vor immer währenden Variablen mit einem beschränkten Geltungsbereich gesetzt.

12.8. Typ-Qualifizierer

Außer der Speicherklasse, die Sie für eine Variable festlegen können, können Sie auch den Typ eines Objekts näher spezifizieren.

12.8.1. volatile

Mit dem Schlüsselwort volatile modifizieren Sie eine Variable so, dass der Wert dieser Variablen vor jedem Zugriff neu aus dem Hauptspeicher eingelesen werden muss.

12.8.2. const

Mit diesem Typ-Qualifizierer definieren Sie eine Konstante. Dies ist eine Variable, deren Wert im Laufe der Programmausführung nicht mehr geändert werden darf.

12.9. Geltungsbereich von Variablen

Die Lebensdauer und der Geltungsbereich von Variablen hängen somit von zwei Punkten ab:

- Von der Position der Deklaration einer Variable
- Vom Speicherklassen-Spezifizierer, der vor einer Variablen steht

Geltungsbereiche:

- Block (block scope): Wird eine Variable in einem Anweisungsblock ({}) deklariert, reichen Geltungsbereich und Lebensdauer dieser Variable vom Anfang des Anweisungsblocks bis zu seinem Ende.
- Funktion (local scope): Wird eine Variable in einer Funktion deklariert, reichen Geltungsbereich und Lebensdauer vom Anfang des Funktionsblocks bis zu seinem Ende. Es sei denn, in der Funktion wird eine Variable innerhalb eines Blocks deklariert. Dann gilt diese Variable nur noch in diesem Block.
- Datei (file scope): Wird eine Variable außerhalb von Funktionen und Anweisungsblöcken deklariert, reichen Geltungsbereich und Lebensdauer vom Punkt der Deklaration bis zum Dateiende.

12.10. Lokale Variablen

Die lokalste Variable ist immer die Variable im Anweisungsblock. Bei gleichnamigen Variablen ist immer die lokalste Variable gültig, also die, die dem Anweisungsblock am nächsten steht.

```
#include <stdio.h>
int main(void) {
   int i=333;
   if(i == 333) {
      int i = 666;
      {
         i = 111;
         printf("%d\n",i);    /* 111 */
      }
      printf("%d\n",i);       /* 111 */
   }
   printf("%d\n",i);          /* 333 */
   return 0;
}
```

Lokale Variablen, die in einem Anweisungsblock definiert wurden, sind außerhalb dieses Anweisungsblocks nicht gültig.

Funktionen sind im Prinzip nichts anderes als Anweisungsblöcke. Sehen Sie sich folgendes Beispiel an:

```
/* func4.c */
#include <stdio.h>
void aendern(void) {
   int i = 111;
   printf("In der Funktion aendern: %d\n",i);
```

```
}
int main(void) {
   int i=333;
   printf("%d\n",i);
   aendern();
   printf("%d\n",i);
   return 0;
}
```

12.11. Globale Variablen

Globale Variablen können Sie sich als Vorwärtsdeklarationen von Funktionen vorstellen. Und wie der Name schon sagt, globale Variablen sind für alle Funktionen gültig. Hier ein Beispiel:

```
/* func5.c */
#include <stdio.h>
int i=333; /* Globale Variable */
void aendern(void) {
   i = 111;
   printf("In der Funktion aendern: %d\n",i); /* 111 */
}
int main(void) {
   printf("%d\n",i);  /* 333 */
   aendern();
   printf("%d\n",i);  /* 111 */
   return 0;
}
```

Natürlich gilt auch hier die Regel, dass bei gleichnamigen Variablen die lokalste Variable den Zuschlag erhält. Beispiel:

```
/* func6.c */
#include <stdio.h>
int i=333;  /* Globale Variable i */
void aendern(void) {
   i = 111;   /* Ändert die globale Variable */
   printf("In der Funktion aendern: %d\n",i);  /* 111 */
}
int main(void) {
   int i = 444;
   printf("%d\n",i);  /* 444 */
   aendern();
   printf("%d\n",i);  /* 444 */
   return 0;
}
```

Merke Für das Anlegen von Variablen gilt, so lokal wie möglich und so global wie nötig.

Tipp Am besten fassen Sie die Definitionen von globalen Variablen in einer zentralen C-Datei zusammen (relativ zum Programm oder Modul) und verwenden externe Deklarationen in Header-Dateien, die dann mit #include eingebunden werden, wo immer Sie die Deklarationen benötigen.

12.12. Statische Variablen

Bevor ich Ihnen die statischen Variablen erklären werde, zunächst folgendes Programmbeispiel:

```
/* func7.c */
#include <stdio.h>
void inkrement(void) {
   int i = 1;
   printf("Wert von i: %d\n",i);
   i++;
}
int main(void) {
   inkrement();
   inkrement();
   inkrement();
   return 0;
}
```

Wenn Sie das Programm ausführen, wird dreimal ausgegeben, dass der »Wert von i: 1« ist. Obwohl man doch hier davon ausgehen muss, dass der Programmierer andere Absichten hatte, da er doch den Wert der Variable i am Ende der Funktion inkrementiert. Dass dies nicht funktioniert, hat mit der Speicherverwaltung des Stack zu tun. Zu diesem Thema in einem anderen Kapitel mehr.

Jetzt können Sie bei diesem Programm vor die Variable das Schlüsselwort static schreiben. Ändern Sie also diese Funktion:

```
void inkrement(void) {
   static int i = 1;
   printf("Wert von i : %d\n",i);
   i++;
}
```

Damit werden für die Variable i tatsächlich die Werte 1, 2 und 3 ausgegeben. Dies haben Sie dem Schlüsselwort static zu verdanken. Denn statische Variablen verlieren bei Beendigung ihres Bezugsrahmens, also bei Beendigung der Funktion, nicht ihren Wert, sondern behalten diesen bei. Dass dies gelingt, liegt daran, dass statische Variablen nicht im Stacksegment der CPU, sondern im Datensegment gespeichert werden. Aber Achtung:

Achtung Statische Variablen müssen schon bei ihrer Deklaration initialisiert werden!

13. Getrenntes Compilieren von Quelldateien

Für kleinere Programme, dessen kompletter Quelltext in einer Datei geschrieben wurde, können Sie beim Kompilieren weiterhin wie bisher verfahren. Bei zunehmend größeren Programmen könnten jedoch folgende Probleme auftreten:

- Editieren und compilieren dauern länger.

- Quelltext wird unübersichtlich und schwieriger zu verstehen.
- Teamarbeit wird erschwert.

Ich werde dieses Beispiel anhand des GNU-Compilers gcc demonstrieren. Mit anderen Compilern dürfte dieser Vorgang ähnlich ablaufen. Bei Entwicklungsumgebungen müssen Sie dabei ein neues Projekt anlegen und die einzelnen Quelldateien zum Projekt hinzufügen. Im Folgenden ein Beispiel zum getrennten Compilieren mit drei Programm-Modulen, die aus Demonstrationszwecken sehr kurz gehalten sind:

```
/*main.c*/
#include <stdio.h>
#include <stdlib.h>
extern void modul1(void);
extern void modul2(void);
int main(void) {
   modul1();
   modul2();
   return EXIT_SUCCESS;
}
/*modul1.c*/
void modul1(void) {
   printf("Ich bin das Modul 1\n");
}
/*modul2.c*/
void modul2(void) {
   printf("Ich bin Modul 2\n");
}
```

Jetzt haben Sie drei Dateien, die zusammen compiliert und gelinkt werden müssen. Zuerst wird dazu der Schalter -c verwendet. Das bewirkt, dass die einzelnen Dateien zwar übersetzt werden, dabei aber nicht der Linkerlauf gestartet wird:

```
gcc -c main.c
gcc -c modul1.c
gcc -c modul2.c
```

Jetzt befinden sich drei Objektdateien (*.obj oder *.o) im Verzeichnis, in dem compiliert wurde. Diese drei Objektdateien können Sie mit folgender Anweisung zu einer ausführbaren Datei linken:

```
gcc main.o modul1.o modul2.o
```

Jetzt befindet sich im Verzeichnis die Datei »a.out«. Dies ist die ausführbare Datei, das fertige Programm. Der Name »a.out« wird standardmäßig vom Compiler verwendet. Mit dem Schalter -o können Sie auch einen eigenen Namen vorgeben:

```
gcc -o myapp main.o modul1.o modul2.o
```

Sollten Sie gcc unter Windows verwenden, fügen Sie beim Programmname myapp die Extension *.exe hinzu (myapp.exe). Wenn Sie jetzt die Datei main.c ändern müssen, brauchen Sie nur diese neu zu übersetzen:

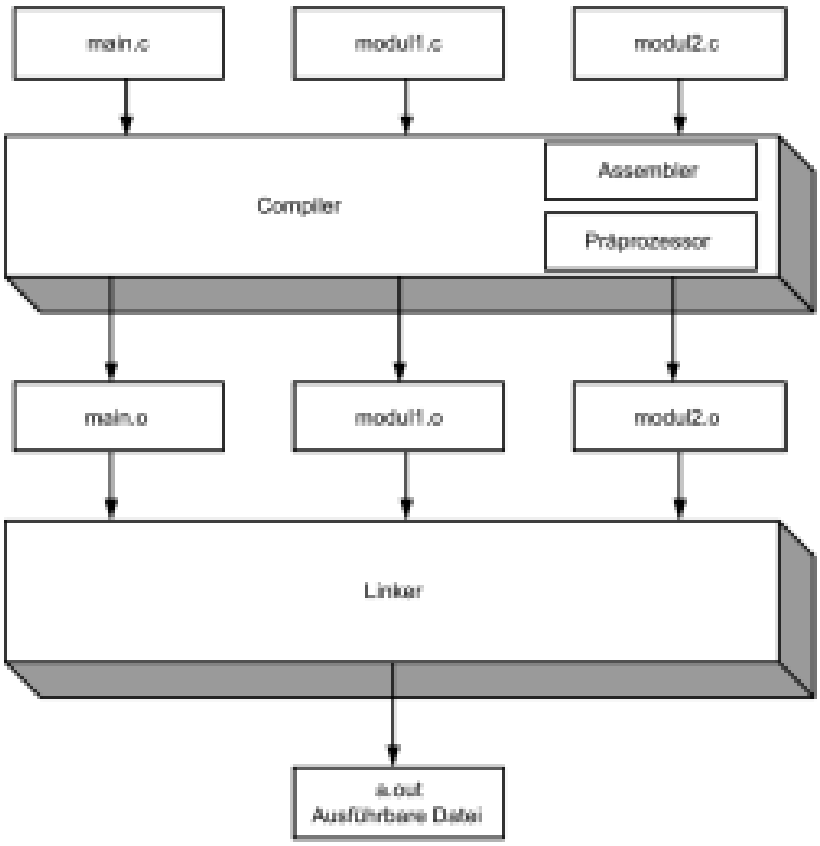
```
gcc -c main.c
```

und können anschließend die einzelnen Dateien wieder zusammenlinken:

```
gcc main.o modul1.o modul2.o
```

Das nochmalige Compilieren der anderen Dateien entfällt, da deren Code nicht geändert wurde und deshalb die unveränderten *.o Dateien neu verlinkt werden können. Bei großen und sehr großen Projekten führt dieses Vorgehen zu deutlich kürzeren Compilier-Zeiten.

Natürlich versteht es sich von selbst, dass sich in einer der Dateien die main()-Funktion befinden muss. Hier der ganze Vorgang noch einmal bildlich:



14. Präprozessor - Direktiven

14.1. #include

Headerdatei	**Bedeutung**
assert.h	Fehlersuche und Debugging
ctype.h	Zeichentest und Konvertierung
errno.h	Fehlercodes
float.h	Limits/Eigenschaften für Gleitpunkttypen

limits.h	Implementierungskonstanten
locale.h	Länderspezifische Eigenschaften
math.h	Mathematische Funktionen
setjmp.h	Unbedingte Sprünge
signal.h	Signale
stdarg.h	Variable Parameterübergabe
stddef.h	Standard-Datentyp
stdio.h	Standard-I/O
stdlib.h	Nützliche Funktionen
string.h	Zeichenkettenoperationen
time.h	Datum und Uhrzeit
conio.h	Consolen-I/O

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

system("clear"); //Bildschirm löschen
```

Die Direktive #include kopiert andere, benannte (Include-) Dateien in das Programm ein. Meistens handelt es sich dabei um Headerdateien mit der Extension *.h oder *.hpp. Hier die Syntax der Präprozessor-Direktive include:

```
#include <header >
#include "header"
```

Der Präprozessor entfernt die include-Zeile und ersetzt diese durch den Quelltext der include-Datei. Der Compiler erhält anschließend einen modifizierten Text zur Übersetzung.

Natürlich können Sie damit eigene Headerdateien schreiben und diese einkopieren lassen. Sie haben beispielsweise eine Headerdatei geschrieben und diese im Verzeichnis /HOME/MYOWNHEADERS unter dem Namen meinheader.h gespeichert. Dann müssen Sie diese Headerdatei am Anfang des Quelltextes mit

```
#include "/home/myownheaders/meinheader.h"
```

einkopieren. Dabei muss dasjenige Verzeichnis angegeben werden, in dem die Headerdatei gespeichert wurde. Steht die Headerdatei hingegen zwischen eckigen Klammern (wie dies bei Standardbibliotheken meistens der Fall ist), also:

```
#include <datei.h>
```

so wird die Headerdatei datei.h im implementierungsdefinierten Pfad gesucht. Dieser Pfad befindet sich in dem Pfad, in dem sich die Headerdateien Ihres Compilers befinden.

Steht die Headerdatei zwischen zwei Hochkommata, also:

```
#include "datei.h"
```

so wird diese im aktuellen Arbeitsverzeichnis oder in dem Verzeichnis gesucht, das mit dem Compiler-Aufruf -I angegeben wurde – vorausgesetzt, Sie übersetzen das Programm in der

Kommandozeile. Sollte diese Suche erfolglos sein, so wird in denselben Pfaden gesucht als wäre #include <datei.h> angegeben.

14.2. #define

Syntax:

```
#define Bezeichner    Ersatzbezeichner

#define EINS 1                // textuelle Ersetzung


z.b:
#define solange   while
#define wenn      if

#define bool      char
#define true      1
#define false     0
```

15. Arrays

15.1. Arrays deklarieren

```
int zahlen[5];
```

15.2. Initialisierung und Zugriff

```
int main(void) {
   int zahlen1[5] = { 0 }; // alle mit Null gefüllt
   int zahlen[] = {3,9,1,2,4,5,8};


   printf("Anz. Elemente : %d\n", sizeof(zahlen) / sizeof(int));
   return EXIT_SUCCESS;
}
```

15.3. Arrays vergleichen

```
int main(void) {
   int i;
   int array1[MAX], array2[MAX];

   for(i = 0; i < MAX; i++) {
         array1[i] = i;
```

```
            array2[i] = i;
        }
        array2[5] = 100; /* array2 an Pos. 5 verändern */

        for(i = 0; i < MAX; i++) {
            if( array1[i] == array2[i] )
                continue;
            else {
                printf("Unterschied an Position %d\n",i);
                break;
            }
        }
        return EXIT_SUCCESS;
    }
```

Achtung: if (array1==array2) … // nur Vergleich von Zeiger

memcmp (array1, array2, sizeof(array1)) // Vergleich von zwei Speierbereichen

15.4. Anzahl der Elemente eines Arrays ermitteln

```
int main(void) {
    int zahlen[] = {3,6,3,5,6,3,8,9,4,2,7,8,9,1,2,4,5};

    printf("Anzahl der Elemente: %d\n", sizeof(zahlen)/sizeof(int));
    return EXIT_SUCCESS;
}
```

15.5. Übergabe von Arrays auf Funktionen

```
void function(int feld[], int n_anzahl) {       // Call by Reference
    int i;

    for(i = 0; i < n_anzahl; i++)
        printf("%d; ", feld[i]);
    printf("\n");
}
```

Aufruf:

```
int zahlen[5];
function(zahlen,5);
function(&zahlen[0], 5);
```

Achtung: Arrays können nicht als Rückgabewert zurückgegeben werden. Man muss sie zuerst in eine Struktur-Typ packen.

15.6. Mehrdimensionale Arrays deklarieren und initialisieren

```
/* 4 Zeilen 5 Spalten */
int Matrix[4][5] = { {10,20,30,40,50},
                     {15,25,35,45,55},
                     {20,30,40,50,60},
                     {25,35,45,55,65}};
```

15.7. Übergabe von mehrdimensionalen Arrays an Funktionen

```
void function(int feld[][DIM2], int dim1)
{
…
}
```

15.8. Arrays in Tabellenkalkulation einlesen

```
/* md_array4.c */
#include <stdio.h>
#include <stdlib.h>
#define WOCHEN  4
#define TAGE    7

float stand[WOCHEN][TAGE] = {
   { 12.3f,13.8f,14.1f,12.2f,15.4f,16.5f,14.3f },
   { 15.4f,13.6f,13.6f,14.6f,15.6f,16.3f,19.5f },
   { 20.5f,20.4f,21.5f,23.4f,21.4f,23.5f,25.7f },
   { 25.5f,26.6f,24.3f,26.5f,26.9f,23.6f,25.4f }
};

int main(void) {
   int i, j;

   printf("Tag;Montag;Dienstag;Mittwoch;Donnerstag; "
          "Freitag;Samstag;Sonntag");
   for(i=0; i < WOCHEN; i++) {
      printf("\nWoche%d;",i);
      for(j=0;j < TAGE; j++) {
         printf("%.2f;",stand[i][j]);
      }
   }
   return EXIT_SUCCESS;
}
```

15.9. Strings/Zeichenketten (char Arrays)

```
/* string1.c */
#include <stdio.h>
#include <stdlib.h>

char hello1[] = { "Hallo Welt\n" };
char output[] = { "Ich bin lesbar \0 Ich nicht mehr" };
char deznu[]  = { "Mich siehst du 0 Mich und die Null auch" };

int main(void) {
   printf("%s",hello1);
   printf("%s\n",output);
   printf("%s\n",deznu);
   return EXIT_SUCCESS;
}
```

```
      char zeile[80] = = { "Hallo Welt\n" };
      zeile[5]='\0';
      zeile[5]=0;
      char zeile[]={`H`,`a`, `l`, `l`, `o`,'\0'};
      char zeile[]={72,97,108,108,111,};
```

Einlesen: scanf(„%s", zeile);
Ausgabe: printf(„%s\n", zeile);

Ausgabe in einen String umleiten: sprintf(zeile2,"Zeileninhalt: %s",zeile);
(Zum Text „Zeileninhalt:" wird die Zeile hinzugefügt und in die Variable „zeile2" gespeichert;)
Von einem String einlesen: sscanf(zeile,"%s %s",wort1, wort2);

15.10.Die Standard Bibliothek <string.h>

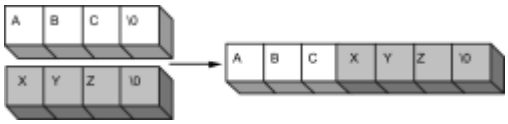
15.10.1. strcat()

```
      char *strcat(char *s1, const char *s2);
```

Um einen String an einen anderen zu hängen, können Sie die Funktion `strcat()` (string catenation) verwenden.

```
char *strcat(char *s1, const char *s2);
```

Damit wird `s2` an das Ende von `s1` angehängt, wobei (logischerweise) das Stringende-Zeichen `'\0'` am Ende von String `s1` überschrieben wird. Voraussetzung ist auch, dass der String `s2` Platz in `s1` hat.



Hier klicken, um das Bild zu Vergrößern

Abbildung 13.14 strcat() – zwei Strings aneinander hängen

Hier ein Beispiel zu `strcat()`:

```
/* stringcat.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char ziel[30] = "Hallo ";
   char name[20];
   printf("Wie heissen Sie: ");
   fgets(name, 20, stdin);
   strcat(ziel, name);
   printf("%s",ziel);
   return EXIT_SUCCESS;
}
```

Hinweis Bitte beachten Sie, dass `strcat()` eine unsichere Funktion ist. Dies deshalb, weil die Länge des Quell-Strings nicht überprüft oder angegeben werden kann. Somit kann mit `strcat()` über den offiziellen Speicherbereich hinaus geschrieben werden. Es wird empfohlen, `strncat()` zu verwenden. Mehr dazu in Abschnitt 25.1, Buffer Overflow

(Speicherüberlauf).

15.10.2. strchr()

```
char *strchr(const char *s, int ch);
```

Wollen Sie in einem String nach einem bestimmten Zeichen suchen, eignet sich die Funktion `strchr()` (string char). Hier die Syntax:

```
char *strchr(const char *s, int ch);
```

Diese Funktion gibt die Position im String `s` beim ersten Auftreten von `ch` zurück. Tritt das Zeichen `ch` nicht auf, wird `NULL` zurückgegeben. Ein Beispiel:

```
/* strchr.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char str[] = "Ein String mit Worten";
   printf("%s\n",strchr(str, (int)'W'));
   return EXIT_SUCCESS;
}
```

Hiermit wird ab Auftreten des Buchstabens `'W'` der komplette String ausgegeben.

15.10.3. strcmp()

```
int strcmp(const char *s1, const char *s2);
```

Für das lexographische Vergleichen zweier Strings kann die Funktion `strcmp()` verwendet werden. Die Syntax lautet:

```
int strcmp(const char *s1, const char *s2);
```

Sind beide Strings identisch, gibt diese Funktion 0 zurück. Ist der String `s1` kleiner als `s2`, ist der Rückgabewert kleiner als 0 und ist `s1` größer als `s2`, dann ist der Rückgabewert größer als 0. Ein Beispiel:

```
/* strcmp.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void String_Vergleich(char s1[], char s2[]) {
   int ret = strcmp (s1, s2);
   if(ret == 0)
      printf("%s == %s\n", s1, s2);
   else
      printf("%s %c %s\n",s1,( (ret < 0) ?'<' :'>'), s2);
}
int main(void) {
   char str1[] = "aaa";
   char str2[] = "aab";
   char str3[] = "abb";
   String_Vergleich(str1, str2);
   String_Vergleich(str1, str3);
   String_Vergleich(str3, str2);
```

```
   String_Vergleich(str1, str1);
   return EXIT_SUCCESS;
}
```

15.10.4. strcpy()

```
char *strcpy(char *s1, const char *s2);
```

Wollen Sie einen String in einen adressierten `char`-Vektor kopieren, können Sie die Funktion `strcpy()` (string copy) nutzen. Die Syntax lautet:

```
char *strcpy(char *s1, const char *s2);
```

Dass hierbei der String-Vektor `s1` groß genug sein muss, versteht sich von selbst. Bitte beachten Sie dabei, dass das Ende-Zeichen `'\0'` auch Platz in `s1` benötigt. Hierzu ein Beispiel:

```
/* strcpy.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char ziel_str[50];
   char str1[] = "Das ist ";
   char str2[] = "ein ";
   char str3[] = "Teststring";
   strcpy(ziel_str, str1);
   /* Ein umständliches Negativbeispiel */
   strcpy(&ziel_str[8], str2);
   /* So ist es einfacher und sicherer */
   strcat(ziel_str, str3);
   printf("%s\n",ziel_str);
   return EXIT_SUCCESS;
}
```

In diesem Beispiel haben Sie gesehen, dass es auch möglich ist, mit `strcpy()` Strings aneinander zu hängen:

```
strcpy(&ziel_str[8], str2);
```

Nur ist das umständlich, und schließlich gibt es dafür die Funktion `strcat()`. Beim Betrachten der Funktion `strcpy()` fällt außerdem auf, dass hierbei ebenfalls nicht überprüft wird, wie viele Zeichen in den Zielstring kopiert werden, womit wieder in einen undefinierten Speicherbereich zugegriffen werden kann. Daher ist auch die Funktion `strcpy()` eine gefährliche Funktion, wenn diese falsch eingesetzt wird. Hierzu sei wieder auf den Abschnitt 25.1 verwiesen.

15.10.5. strspn()

```
int strcspn(const char *s1, const char *s2);
```

Wollen Sie die Länge eines Teilstrings bis zum Auftreten eines bestimmten Zeichens ermitteln, eignet sich die Funktion `strcspn()`. Die Syntax lautet:

```
int strcspn(const char *s1, const char *s2);
```

Sobald ein Zeichen, welches in `s2` angegeben wurde, im String `s1` vorkommt, liefert diese Funktion die Position dazu zurück. Ein Beispiel:

```
/* strcspn.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char string[] = "Das ist ein Teststring";
   int  pos;
   pos = strcspn( string, "Ttg" );
   printf("Erstes Auftreten von T, t oder g an Pos.: %d\n",pos);
   return EXIT_SUCCESS;
}
```

15.10.6. strlen()

```
size_t strlen(const char *s1);
```

Um die Länge eines Strings zu ermitteln, kann die Funktion `strlen()` (string length) eingesetzt werden. Die Syntax lautet:

```
size_t strlen(const char *s1);
```

Damit wird die Länge des adressierten Strings `s1` ohne das abschließende Stringende-Zeichen zurückgegeben. Das Beispiel zu `strlen()`:

```
/* strlen.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char string[] = "Das ist ein Teststring";
   size_t laenge;
   laenge = strlen(string);
   printf("Der String \"%s\" hat %d Zeichen\n",string, laenge);
   return EXIT_SUCCESS;
}
```

Hinweis Dass die Funktion `strlen()` das Stringende-Zeichen `'\0'` nicht mitzählt, ist eine häufige Fehlerquelle, wenn es darum geht, dynamisch Speicher für einen String zu reservieren. Denken Sie daran, dass Sie immer Platz für ein Zeichen mehr bereithalten.

15.10.7. strncat()

```
char *strncat(char *s1, const char *s2, size_t n);
```

Es ist die gleiche Funktion wie `strcat()`, nur dass hiermit `n` Zeichen angehängt werden. Die Syntax lautet:

```
char *strncat(char *s1, const char *s2, size_t n);
```

Diese Funktion ist aus Sicherheitsgründen der Funktion `strcat()` vorzuziehen. Ein Beispiel:

```
/* strncat.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 15
int main(void) {
   char string[MAX] = "Hallo ";
   char puffer[20];
   /* Vorhandenen Platz in string ermitteln */
   size_t len = MAX - strlen(string)+1;
   printf("Ihr Name: ");
   fgets(puffer, 20, stdin);
   strncat(string, puffer, len);
   printf("%s",string);
   return EXIT_SUCCESS;
}
```

Damit ist sichergestellt, dass nicht mehr in einen undefinierten Speicherbereich geschrieben wird.

Hinweis `size_t` ist ein primitiver Datentyp, der meistens als `unsigned int` oder `unsigned long` deklariert ist.

15.10.8. strncmp()

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Diese Funktion arbeitet genauso wie die Funktion `strcmp()`, nur mit dem Unterschied, dass `n` Zeichen miteinander verglichen werden. Die Syntax lautet:

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Hiermit werden also die ersten `n` Zeichen von `s1` und die ersten `n` Zeichen von `s2` lexikographisch miteinander verglichen. Der Rückgabewert ist dabei derselbe wie schon bei `strcmp()`. Ein Beispiel:

```
/* strncmp.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char str1[] = "aaaa";
   char str2[] = "aabb";
   int i;
   for(i = strlen(str1); i > 0; i--) {
      if(strncmp( str1, str2, i) != 0)
         printf("Die ersten %d Zeichen der Strings "
                "sind nicht gleich\n",i);
         else {
            printf("Ab Zeichen %d sind "
                   "beide Strings gleich\n",i);
            /* Weiter vergleich sind nicht mehr nötig */
            break;
         }
   }
   return EXIT_SUCCESS;
}
```

15.10.9. strncpy()

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Die sicherere Alternative zur Funktion `strcpy()` lautet `strncpy()`, welche `n` Zeichen kopiert. Der Ablauf der Funktion ist hingegen wieder derselbe wie bei `strcpy()`. Die Syntax lautet:

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Hier werden `n` Zeichen aus dem String `s2` in den String `s1` ohne das `'\0'`-Zeichen kopiert. Das Beispiel:

```
/* strncpy.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 20
int main(void) {
   char str1[MAX];
   char str2[] = "Ein Teststring welcher laenger"
                 " als 20 Zeichen ist";
   /* MAX-Zeichen in str1 kopieren */
   strncpy(str1, str2, MAX);
   /* Wichtig, String am Ende terminieren !! */
   str1[MAX] = '\0';
   printf("%s\n",str1);
   return EXIT_SUCCESS;
}
```

15.10.10. strpbrk

```
char *strpbrk( const char *s1, const char *s2);
```

Diese Funktion arbeitet ähnlich wie `strcspn()`, nur dass hierbei nicht die Länge eines Teilstrings ermittelt wird, sondern das erste Auftreten eines Zeichens in einem String, welches im Suchstring enthalten ist. Die Syntax lautet:

```
char *strpbrk( const char *s1, const char *s2);
```

Ein Beispiel dazu:

```
/* strpbrk.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char str1[]="Das ist ein Teststring";
   char str2[]="ie";
   printf("%s\n",strpbrk(str1, str2));
   return EXIT_SUCCESS;
}
```

15.10.11. strrchr

```
char *strrchr(const char *s, int ch);
```

Diese Funktion ist der Funktion `strchr()` ähnlich, nur dass hierbei das erste Auftreten des Zeichens von hinten, genauer das letzte, ermittelt wird. Die Syntax lautet:

```
char *strrchr(const char *s, int ch);
```

Die Funktion `fgets()` hängt beim Einlesen eines Strings immer das Newline-Zeichen am Ende an. Manchmal ist das nicht erwünscht. Wir suchen mit `strrchr()` danach und überschreiben diese Position mit dem `'\0'`-Zeichen:

```
/* strrchr.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char string[20];
   char *ptr;
   printf("Eingabe machen: ");
   fgets(string, 20 , stdin);
   /* Zeiger auf die Adresse des Zeichens \n */
   ptr = strrchr(string, '\n');
   /* Zeichen mit \0 überschreiben */
   *ptr = '\0';
   printf("%s",string);
   return EXIT_SUCCESS;
}
```

15.10.12. strspn

```
int strspn(const char *s1, const char *s2);
```

Die Funktion `strspn()` gibt die Position des ersten Auftretens eines Zeichens an, das nicht vorkommt. Die Syntax lautet:

```
int strspn(const char *s1, const char *s2);
```

Ein Beispiel dazu:

```
/* strspn.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char string[] = "75301234-2123";
   int pos = strspn(string, "0123456789");
   printf("Position, welche keine Ziffer ist:");
   printf(" %d\n",pos); /* 8 */
   return EXIT_SUCCESS;
}
```

Dieses Beispiel liefert Ihnen die Position des Zeichens zurück, welches keine Ziffer ist.

15.10.13. strstr()

```
char *strstr(const char *s1, const char *s2);
```

Mit der Funktion `strstr()` können Sie einen String nach Auftreten eines Teilstrings durchsuchen. Die Syntax ist:

```
char *strstr(const char *s1, const char *s2);
```

Damit wird der String `s1` nach einem String mit der Teilfolge `s2` ohne `'\0'` durchsucht. Ein Beispiel:

```
/* strstr.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char string[] = "Das ist ein Teststring";
   char suchstring[] = "ein";
   if( strstr(string, suchstring) != NULL)
      printf("Suchstring \"%s\" gefunden\n", suchstring);
   return EXIT_SUCCESS;
}
```

15.10.14. strtok()

```
char *strtok(char *s1, const char *s2);
```

Mit der Funktion `strtok()` können Sie einen String in einzelne Teilstrings anhand von Tokens zerlegen. Zuerst die Syntax:

```
char *strtok(char *s1, const char *s2);
```

Damit wird der String `s1` durch das Token getrennt, welches sich im `s2` befindet. Ein Token ist ein String, der keine Zeichen aus `s2` enthält. Ein Beispiel:

```
/* strtok.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(void) {
   char string[] = "Ein Teststring mit mehreren Worten\n"
                   "und mehreren Zeilen.\t Ende\n";
   int i=1;
   char *ptr;
   ptr = strtok(string, "\n\t ");
   while(ptr != NULL) {
      printf("% d. Wort: %s\n",i++,ptr);
      ptr = strtok(NULL, "\n\t ");
   }
   return EXIT_SUCCESS;
}
```

Mit der Zeile

```
ptr = strtok(string, "\n\t ");
```

würde nur das erste Wort anhand eines der Whitspace-Zeichen Newline, Tabulator oder Space getrennt werden. Der String wird jetzt von der Funktion `strtok()` zwischengespeichert. Wollen Sie jetzt den String mit weiteren Aufrufen zerlegen, müssen Sie `NULL` verwenden.

```
ptr = strtok(NULL, "\n\t ");
```

Dabei gibt jeder Aufruf das Token zurück. Das jeweilige Trennzeichen wird dabei mit `'\0'` überschrieben. In diesem Beispiel ist die Schleife am Ende, wenn `strtok()` den `NULL`-Zeiger zurückliefert.

Bei CSV-Dateien: `strtok(string, ";");`

16. Strukturen

16.1. Initialisierung und Zugriff auf Strukturen

Bisher wurden mit den Arrays Datenstrukturen desselben Typs verwendet. Bei Strukturen werden jetzt unterschiedliche Datentypen zu einer Struktur zusammengefasst. Anschließend kann auf diese Struktur zugegriffen werden, wie auf einfache Variablen.

Gefolgt von dem Schlüsselwort `struct` wurden alle Daten in einer Struktur namens `adres` zusammengefasst. Die Sichtbarkeit und die Lebensdauer von Strukturen entsprechen exakt der von einfachen Variablen. Der Inhalt der Struktur `adres` wird in geschweiften Klammern zusammengefasst. Am Ende der geschweiften Klammern steht der Variablen-Bezeichner (`adressen`), mit dem auf die Struktur zugegriffen wird.

Zugreifen können Sie auf die einzelnen Variablen einer Struktur mithilfe des Punktoperators (.).

```
/* struct2.c */
#include <stdio.h>
#include <stdlib.h>
#define MAX 30

struct adres {
   char vname[MAX];
   char nname[MAX];
   long PLZ;
   char ort[MAX];
   int geburtsjahr;
} adressen;

oder: struct adres adressen     // Variable anlegen

/*Funktion zur Ausgabe des Satzes*/
void ausgabe(struct adres x) {
   printf("\n\nSie gaben ein:\n\n");
   printf("Vorname.........:%s",   x.vname);
   printf("Nachname........:%s",   x.nname);
   printf("Postleitzahl....:%ld\n",x.PLZ);
   printf("Ort.............:%s",   x.ort);
```

```
   printf("Geburtsjahr.....:%d\n", x.geburtsjahr);
}

int main(void) {
   printf("Vorname      : ");
   scanf(„%s“,&adressen.vname);
        // fgets(adressen.vname, MAX, stdin);
   printf("Nachname     : ");
   fgets(adressen.nname, MAX, stdin);
   printf("Postleitzahl : ");
   do {
      scanf("%5ld",&adressen.PLZ);
   } while(getchar()!= '\n');
   printf("Wohnort      : ");
   fgets(adressen.ort, MAX, stdin);
   printf("Geburtsjahr  : ");
   do {
      scanf("%4ld",&adressen.geburtsjahr);
   } while(getchar()!='\n' );

   ausgabe(adressen);
   return EXIT_SUCCESS;
}
```

Strukturen können natürlich ebenso wie normale Datentypen direkt bei der Deklaration mit Werten initialisiert werden:

```
struct index {
   int seite;
   char titel[30];
} lib = { 308, "Strukturen" };
```

Array von Strukturen:

```
struct adres schueler[3];
schueler[1].PLZ=1234;
```

16.2. Strukturen als Referenzübergabe an Funktionen

Anhand des Funktionsaufrufs vom Beispiel zuvor konnten Sie sehen, dass Strukturen genauso wie jeder andere Datentyp an Funktionen per call-by-value übergeben werden können. Die Funktion bekommt dabei eine Kopie der vollständigen Struktur übergeben. Das Anlegen einer Kopie kann bei häufigen Funktionsaufrufen mit umfangreichen Strukturen die Laufzeit des Programms erheblich beeinträchtigen. Um diesen Mehraufwand zu sparen, empfehle ich Ihnen, Zeiger auf Strukturen als Parameter zu verwenden.

```
/* struct3.c */
#include <stdio.h>
#include <stdlib.h>
```

```
#define MAX 30

struct adres {
   char vname[MAX];
   char nname[MAX];
   long PLZ;
   char ort[MAX];
   int geburtsjahr;
} adressen;

/* Funktion zur Ausgabe des Satzes */
void ausgabe(struct adres *struct_ptr) {
   printf("\n\nSie gaben ein:\n\n");
   printf("Vorname.........:%s",(*struct_ptr).vname);
   printf("Nachname........:%s",(*struct_ptr).nname);
   printf("Postleitzahl....:%ld\n",(*struct_ptr).PLZ);
   printf("Ort.............:%s",(*struct_ptr).ort);
   printf("Geburtsjahr.....:%d\n",(*struct_ptr).geburtsjahr);
}

int main(void) {
   printf("Vorname      : ");
   fgets(adressen.vname, MAX, stdin);
   printf("Nachname     : ");
   fgets(adressen.nname, MAX, stdin);
   printf("Postleitzahl : ");
   do {
      scanf("%5ld",&adressen.PLZ);
   } while(getchar()!= '\n');
   printf("Wohnort      : ");
   fgets(adressen.ort, MAX, stdin);
   printf("Geburtsjahr  : ");
   do {
      scanf("%4ld",&adressen.geburtsjahr);
   } while(getchar()!='\n' );

   ausgabe(&adressen);   // call by reference
   return EXIT_SUCCESS;
}
```

Außer dem Zeiger `struct_ptr` als Parameter, der auf eine Struktur vom Typ `adress` zeigen kann, musste auch der Zugriff auf die Strukturelemente geändert werden. Dass Sie bei Call-by-reference-Variablen mit dem Dereferenzierungsoperator arbeiten müssen, ist Ihnen ja bekannt. Da aber hier der Punkteoperator verwendet wird, muss der Referenzzeiger `struct_ptr` zwischen zwei Klammern gestellt werden, da der Ausdruck zwischen den Klammern die höhere Bindungskraft hat und zuerst ausgewertet wird:

```
printf("Vorname.........:%s",(*struct_ptr).vname);
```

Die Hersteller von C haben aber auch gemerkt, dass eine solche Schreibweise – speziell wenn mehrere Referenzen folgen – schwer lesbar ist. Daher wurde der so genannte Elementkennzeichnungsoperator (`->`) eingeführt. Mit diesem würde die Ausgabe des Vornamens folgendermaßen vorgenommen werden:

```
printf("Vorname.........:%s", struct_ptr->vname;
```

16.3. Strukturen als Rückgabewert einer Funktion

Wie bei der Werteübergabe von Strukturen, sollten Sie auch bei der Werterückgabe von Strukturen Zeiger verwenden. Dafür soll wieder das eben geschriebene Listing verwendet werden. Es fehlt nämlich noch eine Funktion zur Eingabe der einzelnen Strukturelemente. Hier das Listing mit der Funktion `eingabe()`, welche die Anfangsadresse der Struktur zurückgibt:

```
/* struct4.c */
#include <stdio.h>
#include <stdlib.h>
#define MAX 30
struct adres {
   char vname[MAX];
   char nname[MAX];
   long PLZ;
   char ort[MAX];
   int geburtsjahr;
};
/* Funktion zur Ausgabe des Satzes */
void ausgabe(struct adres *struct_ptr) {
   printf("\n\nSie gaben ein:\n\n");
   printf("Vorname.........:%s",struct_ptr->vname);
   printf("Nachname........:%s",struct_ptr->nname);
   printf("Postleitzahl....:%ld\n",struct_ptr->PLZ);
   printf("Ort.............:%s",struct_ptr->ort);
   printf("Geburtsjahr.....:%d\n",struct_ptr->geburtsjahr);
}
struct adres *eingabe(void) {
   static struct adres *adressen;
   adressen = (struct adres *)malloc(sizeof(struct adres));
   printf("Vorname : ");
   fgets(adressen->vname, MAX, stdin);
   printf("Nachname : ");
   fgets(adressen->nname, MAX, stdin);
   printf("Postleitzahl : ");
   do {scanf("%ld",&adressen->PLZ);} while(getchar()!= '\n');
   printf("Wohnort : ");
   fgets(adressen->ort, MAX, stdin);
   printf("Geburtsjahr : ");
   do {
      scanf("%ld",&adressen->geburtsjahr);
   }while(getchar()!='\n' );
   return adressen;
}
int main(void) {
   struct adres *adresse1, *adresse2;
   adresse1=eingabe();
   adresse2=eingabe();
   ausgabe(adresse1);
   ausgabe(adresse2);
   return EXIT_SUCCESS;
}
```

Bei diesem Listing verwenden Sie bereits nur noch Zeiger und kommen zum ersten Mal mit den dynamischen Datenstrukturen in Berührung. Aufgerufen wird diese Funktion zur Eingabe von Strukturelementen mit:

```
adresse1 = eingabe();
```

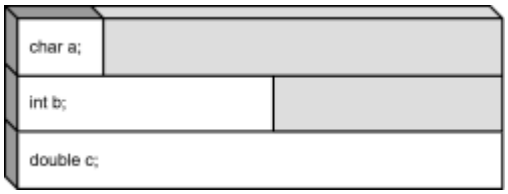
Einen Adressoperator benötigen Sie hier nicht, da Sie ja bereits einen Zeiger verwenden. Und da Sie hier einen Zeiger verwenden, benötigen Sie in der Funktion erst einmal Speicherplatz für eine Struktur. Dieser wird mit der Funktion `malloc()` angefordert:

```
static struct adres *adressen;
adressen = (struct adres *)malloc(sizeof(struct adres));
```

16.4. Union

Eine weitere Möglichkeit, Daten zu strukturieren, sind Unions (auch Varianten genannt). Abgesehen von einem anderen Schlüsselwort, bestehen zwischen Unions und Strukturen keine syntaktischen Unterschiede. Der Unterschied liegt in der Art und Weise, wie mit dem Speicherplatz der Daten umgegangen wird. Hierzu ein Beispiel der Speicherplatzbelegung einer Struktur:

```
struct test1 {
   char a;
   int b;
   double c;
};
```



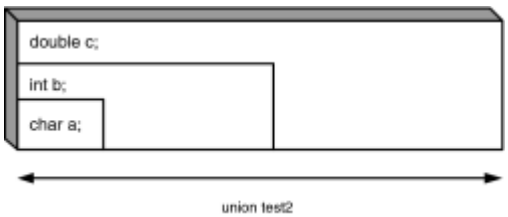
Hier klicken, um das Bild zu Vergrößern

Abbildung 17.8 Speicherbelegung bei einer Struktur

Diese Struktur benötigt 16 Byte an Speicher. Das gleiche Beispiel jetzt mit dem Schlüsselwort `union`:

```
union test2 {
   char a;
   int b;
   double c;
};
```

Intern sieht dies mit `union` folgendermaßen aus:



Hier klicken, um das Bild zu Vergrößern

Abbildung 17.9 Speicherbelegung einer Struktur mit dem Schlüsselwort union

```
/* union1.c */
#include <stdio.h>
#include <stdlib.h>

struct test1 {
   char a;
   int b;
   double c;
};

union test2 {
   char a;
   int b;
   double c;
};

int main(void) {
   printf("struct benoetigt %d Bytes\n", sizeof(struct test1));
   printf("Union  benoetigt %d Bytes\n", sizeof(union test2));
   return EXIT_SUCCESS;
}
```

Die Struktur mit dem Schlüsselwort `union` besitzt jetzt nur noch acht Bytes?! Dies verursacht das Schlüsselwort `union` und das größte Element in der Struktur, `double`. Dieser ersparte Speicherplatz wird allerdings mit dem Nachteil erkauft, dass immer nur ein Element in dieser Struktur verwendet werden kann. Beispielsweise wird dem Strukturelement `int b` der Wert 100 übergeben:

```
text.b = 100;
```

Somit beträgt die Speichergröße der Struktur trotzdem acht Bytes für einen Integer, da der größte Datentyp in der `union` nämlich `double` lautet. Übergeben Sie jetzt an `double c` den Wert 10.55:

```
text.c = 10.55;
```

Jetzt können Sie auf den Wert von `int b` nicht mehr zugreifen, da dieser von `double c` überlappt wurde. Zwar kann es immer noch sein, dass `int b` weiterhin den Wert 100 ausgibt, aber dies wäre Zufall, denn der Speicherbereich wurde überdeckt. Das Verhalten ist in diesem Fall undefiniert.

Welchen Vorteil hat es, wenn immer auf ein Element einer Struktur zugegriffen werden kann? Der wesentliche Vorteil liegt in der Anwendung von Union zum Einsparen von Speicherplatz bei der Verarbeitung großer Strukturen; beispielsweise bei Strukturen, wo bestimmte Elemente niemals miteinander auftreten. Ein Computerspiel z.B., bei dem sich immer einer gegen einen, Auge um Auge, duelliert. In dem folgenden Codeabschnitt wurde eine Struktur mit vier verschiedenen Gegnercharakteren erstellt:

16.5. Aufzählungstyp enum

```
/* enum1.c */
#include <stdio.h>
#include <stdlib.h>

enum zahl { NU_LL, EINS, ZWEI, DREI, VIER};

int main(void) {
   enum zahl x;
   x=NU_LL;
   printf("%d\n",x);

   x=EINS;
   printf("%d\n",x);

   x=ZWEI;
   printf("%d\n",x);

   x=DREI;
   printf("%d\n",x);

   x=VIER;
   printf("%d\n",x);
   return EXIT_SUCCESS;
}
```

Bei Ausführung des Programms werden die Zahlen von null bis vier auf dem Bildschirm ausgegeben. Die Aufzählung lautet hier:

```
enum zahl { NU_LL, EINS, ZWEI, DREI, VIER };
```

In der Regel beginnt der Aufzählungstyp, sofern nicht anders angegeben, mit 0; also `NU_LL=0`. Das nächste Feld, wenn nicht anders angegeben, hat den Wert 1. Somit ist `EINS` auch 1. Gleichbedeutend hätte man dies auch so schreiben können:

```
enum zahl { NU_LL=0, EINS=1 ,ZWEI=2 ,DREI=3 ,VIER=4 };
```

Wird `enum` hingegen so benutzt:

```
enum farben { rot, gelb=6, blau, gruen };
```

würden folgende Konstanten definiert werden:

```
enum farben { 0, 6, 7, 8 };
```

Die Farbe `gelb` wurde mit dem Wert 6 initialisiert. Die Steigerung des Werts zur nächsten Konstante beträgt bei `enum` immer plus eins. Somit hat die Konstante `blau` den Wert 7 und `gruen` den Wert 8.

Der Aufzählungstyp `enum` dient der besseren Lesbarkeit eines Programms. `BOOL` (in C++ gibt es diesen Datentyp wirklich) könnten Sie aber auch als Makro implementieren:

```
#define BOOL int
#define FALSE 0
#define TRUE  1
```

Wo liegt dann der Unterschied zwischen `enum` und einer Reihe von Präprozessor-Defines? Ironischerweise besteht kaum ein Unterschied. Geplant war (laut ANSI C-Standard) `enum`, um ohne Casts verschiedene integrale Typen vermischen zu können, was ja sonst in der Regel einen Compilerfehler zur Folge hat. So hätten eine Menge Programmierfehler aufgefangen werden können.

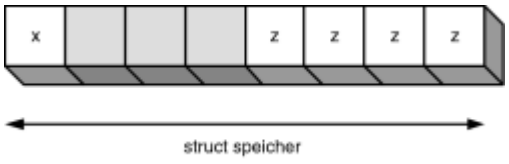
Aber enum hat auch Vorteile:
Zahlenwerte werden automatisch zugewiesen.
Debugger-Werte von enum-Variablen können symbolisch dargestellt werden
enum unterliegt auch der Sichtbarkeitsregel von C.

16.6. Attribute von Strukturen verändern

Der Speicherplatz für eine Struktur wird – wie schon bei den Arrays – lückenlos im Hauptspeicher abgelegt. Damit das System schneller auf diese Daten im Hauptspeicher zurückgreifen kann, werden diese in zwei oder in durch vier teilbare Adressen angeordnet. Dies ist abhängig vom Betriebssystem. Dabei wird einiges an Speicherplatz verschwendet. Zur Demonstration dient das folgende Programm:

```
/* alignment1.c */
#include <stdio.h>
#include <stdlib.h>
struct speicher {
   char x;
   int z;
};
int main(void) {
   struct speicher test;
   printf("%ld Bytes\n",sizeof(test));
   return EXIT_SUCCESS;
}
```

Auf 32-Bit-Systemen dürfte diese Struktur acht Byte benötigen. Und dies, obwohl es eigentlich fünf Byte sein sollten (`char` + `int` = 5 Byte).



Hier klicken, um das Bild zu Vergrößern

Abbildung 17.10 Speicherabbild mit einer Struktur mit unbenannten Lücken

Diese Abbildung stellt ein Vier-Byte-Alignment dar, wie es bei den meisten Systemen der Fall sein wird. Dabei entsteht eine Lücke von drei Byte (grau gefärbt), welche ungenutzt bleibt. Es wird hier auch vom »Padding« (Auffüllen, Polsterung) des Speichers gesprochen.

Hinweis Dies ist übrigens auch der Grund, warum Sie Strukturen nicht direkt

miteinander vergleichen können. Auch ein Low-Level-Vergleich, Byte für Byte, kann da nicht viel helfen, da dieser durch zufällig gesetzte Bits in den Löchern verfälscht sein könnte. In solch einem Fall müssen Sie sich mit einer eigenen Funktion, welche die einzelnen Strukturelemente miteinander vergleicht, selbst behelfen.

Viele Compiler besitzen daher einen speziellen Schalter, mit dem diese Lücke entfernt werden kann. Wobei ich gleich anmerken muss, dass dies nicht ANSI C-konform und Compiler-abhängig ist.

Mit dem Schalter

```
__attribut__
```

können dem Compiler mehrere Informationen zu einer Funktion, zu Variablen oder Datentypen übergeben werden. Um damit eine lückenlose Speicherbelegung zu erreichen, könnten Sie das Attribut `packed` verwenden.

Sollte dieser Schalter bei Ihrem Compiler nicht funktionieren, können Sie auch das Pragma `pack` verwenden:

```
#pragma pack(n)
```

Für `n` kann hier der Wert 1, 2, 4, 8 oder 16 angegeben werden. Je nachdem, welche Angabe Sie dabei machen, wird jedes Strukturelement nach dem ersten kleineren Elementtyp oder auf `n` Byte abgespeichert.

Beim Testen auf verschiedenen Systemen und unterschiedlichen Compilern gab es keine Probleme mit dem `#pragma pack`. Die Option `__attribut__` hingegen wurde nicht von jedem Compiler erkannt. Wie Sie dabei vorgehen, müssen Sie letztendlich selbst herausfinden.

Hier das Beispiel dazu:

```
/* alignment2.c */
#include <stdio.h>
#include <stdlib.h>
/* Lässt sich dieses Listing nicht übersetzen,
 * entfernen Sie __attribute__((packed)) und
 * verwenden stattdessen #pragma pack(1)
 */
/* #pragma pack(1) */
struct speicher {
   char x;
   int z;
} __attribute__ ((packed));
int main(void) {
   struct speicher test;
   printf("%ld Bytes\n",sizeof(test));
   return EXIT_SUCCESS;
}
```

17. Ein-/Ausgabefunktionen

17.1. Höhere Ein-/Ausgabefunktionen (in der Headerdatei `<stdio.h>`)

Funktion	Beschreibung
fopen	Datenstrom öffnen
fclose	Datenstrom schließen
feof	Testet auf Dateiende im Stream
ferror	Testet auf Fehler im Stream
fflush	Leert den Puffer im Datenstrom
fgetc	Zeichenweise lesen vom Stream
fgets	Zeilenweise lesen vom Stream
fgetpos	Position im Stream ermitteln
fprintf	Formatierte Ausgabe an Stream
fputc	Zeichenweise schreiben in den Stream
fputs	Schreibt einen String in den Stream
freopen	Datenstrom erneut öffnen
fscanf	Formatierte Eingabe vom Stream
fseek	Dateizeiger neu positionieren
fsetpos	Dateizeiger neu positionieren
ftell	Position im Stream ermitteln
getc	Zeichenweise lesen vom Stream
getchar	Zeichenweise lesen von stdin
gets	Liest String von stdin (unsichereFunktion)
printf	Formatierte Ausgabe an stdout
putc	Zeichenweise schreiben in den Stream
putchar	Zeichenweise schreiben an stdout
puts	Zeichenkette an Stream stdout
rewind	Position von Stream auf Anfang
scanf	Formatierte Eingabe von stdin
setbuf	Streampuffer einrichten
setvbuf	Streampuffer verändern
sprintf	Formatierte Ausgabe in einem String
sscanf	Formatierte Eingabe aus einem String
ungetc	Zeichen zurück in den Stream

17.2. Zeichenweise Lesen und Schreiben

Um zeichenweise aus dem Stream `stdin` (Standardeingabe) zu lesen und zeichenweise auf `stdout` (Standardausgabe) zu schreiben, können folgende Funktionen verwendet werden: Befehl: c = getchar()) und putchar(c);

```
/* echo_char.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int c;
   while( (c = getchar()) !='.')
      putchar(c);
   return EXIT_SUCCESS;
}
```

```
Weitere Anweisung: while((c = getchar()) != EOF);
```

Hiermit werden solange Zeichen eingelesen, bis die Tastenkombination **(Strg)** + **(Z)** (unter MS-Systemen) oder **(Strg)** + **(D)** (unter Linux) gedrückt wird, welche `EOF` nachbildet.

```
/* count_char.c */
#include <stdio.h>
#include <stdlib.h>

int main (void) {
   int c,counter=0;

   printf("Bitte Eingabe machen:");
   /* Eingabe machen bis mit Return beendet wird */
   while((c=getchar()) != '\n') {
      /* Leerzeichen und Tabulatorzeichen nicht mitzählen */
      if( (c != ' ') && (c != '\t') )
         counter++;     /* counter erhöhen */
   }
   /* Gibt die Anzahl eingegeb. Zeichen von 0 bis counter-1 aus
    * mit counter-1 wird das Zeichen '\0' nicht mitgezählt */
   printf("Anzahl der Zeichen beträgt %d Zeichen\n", counter-1);
   return EXIT_SUCCESS;
}
```

Hinweis für Programmierneulinge Auch wenn es bei der Funktion `getchar()` den Anschein hat, dass hier mit ganzen Strings gearbeitet wird, ist dem nicht so. Diese Funktion liest Zeichen für Zeichen aus einem Puffer.

Dies geschieht aber erst, wenn die Taste ENTER gedrückt wird. Suchen Sie nach einer Funktion, die auf Betätigung einer bestimmten Taste wartet, könnte die Funktion `getch()` für Sie interessant sein. Für MS-DOS steht diese Funktion sofort zur Verfügung, z.B.: `while( (c=getch()) != 'q');`. Damit wird das Programm so lange angehalten, bis die Taste `q` gedrückt wird. Unter Linux müssen Sie dafür die Bibliothek `<ncurses.h>` oder `<termios.h>` verwenden. Der Nachteil von `getch()` ist, dass das Programm damit schlecht auf ein anderes System portiert werden kann.

17.3. Datei/Stream öffnen (-fopen)

Zuerst soll eine einfache Textdatei zum Lesen geöffnet werden. Dabei wird folgendermaßen vorgegangen:

Zuerst soll eine einfache Textdatei zum Lesen geöffnet werden

```
FILE *datei;
...
datei = fopen("textdatei.txt", "r");
```

Konnte diese Datei nicht geöffnet werden bzw. ist sie nicht vorhanden, dann liefert die Funktion `fopen()` den `NULL`-Zeiger zurück.

Bei Microsoft-Systemen muss darauf geachtet werden, dass statt nur einem Backslash zwei (`\\`) geschrieben werden, um das Zeichen `'\'` anzuzeigen. Bei Linux/UNIX ist das einfacher. Ist das Verzeichnis folgendes `/home/Texte/test.txt`

Modus	Bedeutung
"r"	Öffnen einer Datei zum Lesen. Wenn die Datei nicht existiert oder nicht geöffnet werden konnte, gibt fopen() NULL zurück.
"w"	Anlegen einer Datei zum Ändern. Wenn die Datei nicht geändert werden kann bzw. wenn keine Schreibberechtigung besteht, liefert hier fopen() NULL zurück. Wenn unter Windows/MS-Dos die Datei ein Readonly-Attribut hat, kann diese nicht geöffnet werden.
"a"	Öffnet die Datei zum Schreiben oder Anhängen ans Ende der Datei. Wenn die Datei nicht vorhanden ist, liefert fopen() wieder NULL zurück. Auch NULL wird zurückgeliefert, wenn keine Zugriffsrechte bestehen.
"r+"	Öffnet die Datei zum Lesen und Schreiben, also zum Verändern. Bei Fehlern oder mangelnden Rechten liefert fopen() auch hier NULL zurück.
"w+"	Anlegen einer Datei zum Ändern. Existiert eine Datei mit gleichem Namen, wird diese zuvor gelöscht. Bei Fehlern oder mangelnden Rechten liefert fopen() hier NULL zurück.
"a+"	Öffnen einer Datei zum Lesen oder Schreiben am Ende der Datei bzw. die Datei wird angelegt, falls noch nicht vorhanden. Bei Fehlern oder mangelnden Rechten liefert fopen() NULL zurück.

Bewirkt	r	w	a	r+	w+	a+
Datei ist lesbar	x			x	x	x
Datei ist beschreibbar		x	x	x	x	x
Datei ist nur am Dateiende beschreibbar			x			x
Existierender Dateiinhalt geht verloren		x			x	

Zusätzlicher Modus	Bedeutung
b	Die Datei wird im Binärmodus geöffnet. Die Zeichen werden dabei nicht verändert bzw. konvertiert. Das heißt, jedes Zeichen wird so weitergegeben, wie es in der Datei steht, und es wird so in die Datei geschrieben, wie die Schreibfunktion eingestellt ist. Der Modus b wird bei Linux nicht verwendet und bei Angabe ignoriert. Er wird nur aus Kompatibilitätsgründen zu ANSI C erhalten.
t	Die Datei wird im Textmodus geöffnet und sollte daher auch lesbare Textzeichen beinhalten.

```
#include <stdio.h>
FILE *fopen(const char *pfadname, const char *modus);
```

```
/* fopen1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   FILE *datei;

   /* Bitte Pfad und Dateinamen anpassen */
   datei = fopen("test.txt", "r");
   if(NULL == datei) {
      printf("Konnte Datei \"test.txt\" nicht öffnen!\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

17.4. Zeichenweise Lesen und Schreiben – putc/fputc und getc/fgetc

Die Funktionen `getc()` und `fgetc()` sind das dateiorientierte Gegenstück zu `getchar()`. Sie werden verwendet, um einzelne Zeichen aus einem Stream zu lesen, der zuvor mit `fopen()` geöffnet wurde. Der Unterschied zwischen `getc()` und `fgetc()` liegt darin, dass `fgetc()` als eine Funktion implementiert ist und `getc()` ein Makro sein darf. Hier die Syntax dazu:

```
#include <stdio.h>
int getc(FILE *datei);
int fgetc(FILE *datei);
```

Folgende beiden Schreibweisen sind dabei identisch:

```
// liest ein Zeichen aus der Standardeingabe
getchar();
// liest ebenfalls ein Zeichen aus der Standardeingabe
fgetc(stdin);
```

```
* fgetc1.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int c;
   FILE *datei;
   datei=fopen("test.txt", "r");
   if(datei != NULL) {
      while( (c=fgetc(datei)) != EOF)
         putchar(c);
   }
   else {
      printf("Konnte Datei nicht finden bzw. öffnen!\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

Damit wird das Zeichen `quelle` in den Stream `ziel` geschrieben.

```
  FILE *quelle, *ziel;
  int c;

  quelle=fopen(name_q,"rb");
  ziel=fopen(name_z,"w+b");

  /*Wir kopieren zeichenweise von quelle nach ziel */
  while( (c=getc(quelle)) != EOF)
            putc(c,ziel);
```

17.5. Formatiertes Einlesen/Ausgeben von Streams mit fprintf und fscanf

```
fscanf(stdin, "%d", &x);
fprintf(stdout, "Hallo Welt\n");
fprintf(dateiname, "Hallo Welt\n");
```

17.6. Stream positionieren – fseek

```
#include <stdio.h>
int fseek(FILE *datei, long offset, int origin);
```

Mit `fseek()` kann der Schreib-/Lesezeiger des Streams `datei` verschoben werden. Die Positionierung wird mit `offset` und `origin` angegeben. `origin` gibt den Bezugspunkt an, von wo ab der Schreib-/Lesezeiger verschoben werden soll. `offset` gibt an, wie weit von diesem Bezugspunkt aus der Dateizeiger verschoben wird. Für `origin` sind drei symbolische Konstanten in der Headerdatei `<stdio.h>` deklariert:

Symbol	Wert	Offset-Rechnung ab
SEEK_SET	0	Anfang der Datei
SEEK_CUR	1	Aktuelle Position

SEEK_END	2	Ende der Datei

```
/* Sie geben die ganze Datei auf dem Bildschirm aus */
  fseek(quelle, 0L, SEEK_SET);
/* Die letzten 10 Zeichen einer Datei ausgeben*/
  fseek(quelle, -10L, SEEK_END);
```

17.7. Zeilenweise Lesen mit gets/fgets

Nun folgen Funktionen zum zeilenweisen Lesen und Schreiben von einem oder in einen Stream. Zuerst die Funktionen zum Lesen:

```
#include <stdio.h>
char *gets(char *puffer);
char *fgets(char *puffer, int n, FILE *datei);
```

Mit `fgets()` wird zeilenweise vom Stream `datei` bis zum nächsten Newline-Zeichen gelesen. Die gelesene Zeile befindet sich in der Adresse von `puffer` mit dem Newline-Zeichen `'\n'` und dem abschließenden `'\0'`-Zeichen. Mit `gets()` können Sie ebenso zeilenweise einlesen, allerdings nur von der Standardeingabe (`stdin`). Beispielsweise:

```
/* gets.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   char name[20];
   printf("Bitte geben Sie Ihren Namen ein : ");
   gets(name);  /* Gefährlich */
   printf("Hallo %s\n",name);
   return EXIT_SUCCESS;
}
```

Da die Funktion `gets()` nicht die Anzahl der einzugebenden Zeichen überprüft, kann dies zu einem Pufferüberlauf (Buffer Overflow) führen. Deshalb sollten Sie auf keinen Fall `gets()`, sondern die Funktion `fgets()` verwenden.

Wenn Sie die Syntax von `fgets()` betrachten, bemerken Sie, dass sich darin außer der Zieladresse, in der die Daten eingelesen werden, zusätzlich ein Stream (`FILE` Zeiger) und ein Integer-Wert befinden, der die Anzahl der einzulesenden Zeichen festlegt. Mit `fgets` werden somit `n` Zeichen oder bis zum nächsten Newline (`'\n'`) aus dem Stream in die Adresse von `puffer` gelesen. Wobei der Stream eine beliebig geöffnete Datei oder auch die Standardeingabe (`stdin`) sein kann. Hierzu das vorige Beispiel mit `fgets()`:

```
/* fgets1.c */
#include <stdio.h>
#include <stdlib.h>
#define MAX 20
int main(void) {
   char name[MAX];
   printf("Bitte geben Sie Ihren Namen ein : ");
   fgets(name, MAX, stdin);
   printf("Hallo %s",name);
   return EXIT_SUCCESS;
}
```

Sollten hier mehr als 20 Zeichen eingegeben werden, läuft das Programm trotzdem für immer anstandslos. Es werden 20 Zeichen bzw. 18 darstellbare Zeichen + `'\n'` + `'\0'` an den String

name übergeben. Ein Vorteil ist, dass mit fgets() nicht nur von stdin gelesen werden kann, sondern auch von einem beliebigen Stream. Hier ein Beispiel, wie Sie mit fgets() zeilenweise aus einer Datei lesen können

```
#include <stdio.h>
#include <stdlib.h>
#define ZEILENLAENGE 80
int main(void) {
   FILE *quelle;
   char puffer[ZEILENLAENGE], name[20];
   printf("Welche Datei wollen Sie zum Lesen öffnen: ");
   scanf("%s",name);
   if( (quelle=fopen(name,"r")) == NULL) {
      fprintf(stderr, "Kann %s nicht oeffnen\n", name);
      return EXIT_FAILURE;
   }
   while(fgets(puffer, ZEILENLAENGE, quelle))
      fputs(puffer, stdout);
   return EXIT_SUCCESS;
}
```

17.8. Zeilenweise Schreiben mit puts/fput

Mit puts() wird eine ganze Zeile auf dem Bildschirm (stdout) ausgegeben. Außerdem gibt puts() am Ende der Zeichenkette noch ein '\n'-Zeichen mit aus, die Funktion fputs() macht dies hingegen nicht. Im Gegensatz zu puts(), womit Sie nur auf die Standardausgabe (stdout) schreiben können, verwendet fputs(), wie schon fgets(), einen beliebig offenen Stream, in den geschrieben wird. Als Stream ist eine Datei zulässig, die mit einem Schreibmodus geöffnet wurde, oder auch die Standardausgabe (stdout). Hier die Syntax der beiden Funktionen:

```
#include <stdio.h>
int puts(const char *puffer);
int fputs(const char *puffer, FILE *datei);
```

```
FILE *quelle, *kopie;
quelle=fopen(name,"r");
kopie=fopen("kopie.txt","w");
  while(fgets(puffer,ZEILENLAENGE,quelle)) {
      fputs(puffer, kopie);
      puts(puffer);
   }
```

17.9. Blockweise Lesen und Schreiben – fread und fwrite

Die Funktionen fread() und fwrite() erledigen das binäres Lesen und Schreiben ganzer Blöcke (Strukturen).

```
size_t fread(void *puffer, size_t blockgroesse,
             size_t blockzahl, FILE *datei);
```

z.B:

```
   int puffer[10];
   FILE *quelle;
   int i;
   quelle = fopen("wert.dat", "r+b");
   if(quelle != NULL)
      fread(&puffer, sizeof(int), 10, quelle);
```

`fread()` liest `10` Datenobjekte mit der Größe von je `sizeof(int)` Bytes aus dem Stream `quelle` in die Adresse von `puffer (= Array)`.

Jetzt zur Funktion `fwrite()`:

```
size_t fwrite(const void *puffer,size_t blockgroesse,
              size_t blockzahl, FILE *datei);
```

```
struct {
   char name[20];
   char vornam[20];
   char wohnort[30];
   int alter;
   int plz;
   char Strasse[30];
} adressen;
FILE *quelle;
if((quelle=fopen("adress.dat", "w+b")) == NULL)
...
```
fwrite(&adressen, sizeof(struct adressen), 1, quelle);

17.10.Datei schließen (-fclose)

fclose(quell);

17.11.Datei löschen oder umbenennen

```
remove(dateiname);
```
rename(alt,neu);

z.B:
```
   char dateiname[100];

   printf("Welche Datei wollen Sie löschen?\n");
   printf("Bitte wenn nötig gültigen Pfad angeben.\n");
   printf("Eingabe :> ");
   scanf("%99s",dateiname);
   if((remove(dateiname)) < 0) {
      fprintf(stderr, "Fehler beim Löschen von %s", dateiname);
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

17.12. Formatiert in einem String schreiben und formatiert aus einem String lesen – sscanf und sprintf

```
char string1[80];
sprintf(string1, "%d  %d",wert1,zahl2);
sprintf(string1, "%s %d",&zahl2, &wert1);
```

18. Attribute von Dateien und Arbeiten mit Verzeichnissen

18.1. Attribute einer Datei ermitteln - stat()

Der Funktion stat()wird die Adresse der Struktur struct stat übergeben. Aus dieser Struktur können die Attribute der Datei ausgelesen werden. Hierzu die Syntax:

```
#include <sys/stat.h>    /* LINUX/UNIX */
#include <sys/types.h>   /* LINUX/UNIX     */
#include <sys\stat.h>    /* MS-DOS/WINDOWS */

int stat(const char *pfad, struct stat *puffer);
```

Mit stat() werden somit die Attribute der Datei ,welche Sie mit pfad angeben, in die Adresse der Strukturvariable puffer geschrieben. Ein Beispiel:

```
struct stat attribut;
stat("testprogramm.txt", &attribut);
...
if(attribut.st_mode & S_IFCHR)
   printf("Datei ist eine Gerätedatei");
```

18.2. Prüfen des Zugriffsrechts – access

Mit der Funktion access() können Sie feststellen, ob ein Prozess bestimmte Berechtigungen für den Zugriff auf eine Datei hat. Die Syntax zur Funktion access() lautet:

```
#include <unistd.h>  /* für UNIX/LINUX */
#include <io.h>      /* für MS-DOS     */

int access(const char *pfad, int modus);
```

So wird überprüft, ob der pfad der Datei existiert und die Zugriffsrechte modus besitzt. Folgende Zugriffsrechte (Modus) existieren:

Modus	Bedeutung
00 oder F_OK	Datei existiert
01 oder X_OK	Datei ausführbar (nur Linux/UNIX)
02 oder W_OK	Datei beschreibbar
04 oder R_OK	Datei lesbar
06 oder W_OK|R_OK	Datei lesbar und beschreibbar

18.3. Verzeichnis-Funktionen

```
/* create_dir.c */
#ifdef __unix__
   #include <sys/types.h>
   #include <sys/stat.h>
   #define MODUS ,0711)
#elif __WIN32__ || _MS_DOS_
    #include <dir.h>
    #define MODUS )
#else
    #include <direct.h>  /* Visual C++ */
    #define MODUS )
#endif
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   char pfadname[200];

   printf("Wie soll der neue Ordner heissen: ");
   scanf("%199s",pfadname);
   if(mkdir(pfadname MODUS == -1) /*Nicht schön, aber portabler*/
      printf("Konnte kein neues Verzeichnis erstellen\n");
   else
      printf("Neues Verzeichnis namens %s erstellt\n",pfadname);
   return EXIT_SUCCESS;
}
```

19. Kapitel 15 Kommandozeilenargumente

.

19.1. 15.1 Argumente an die Hauptfunktion

Um einem Programm beim Start Argumente zu übergeben, wird eine parametrisierte Hauptfunktion benötigt. Hierzu die Syntax:

```
int main(int argc, char **argv)
```

Diese Hauptfunktion main() besitzt zwei Parameter mit den Namen argc und argv. Die Namen dieser Parameter sind so nicht vorgeschrieben. Sie können genauso gut schreiben:

Der erste Parameter beinhaltet die Anzahl von Argumenten, welche dem Programm beim Start übergeben wurden. Dabei handelt es sich um einen Integerwert. Im zweiten Parameter stehen die einzelnen Argumente. Diese werden als Strings in einer Stringtabelle gespeichert. Folgendes Beispiel demonstriert dies:

```
/* argument.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
   int i;
   for(i=0; i < argc; i++) {
      printf("argv[%d] = %s ", i, argv[i]);
      printf("\n");
   }
   return EXIT_SUCCESS;
}
```

Das Listing wurde z.B. unter dem Namen argument.c gespeichert und anschließend übersetzt. Starten Sie das Programm, wird auf dem Bildschirm der Programmname ausgegeben:

Aufruf: `argument Hallo Welt`

Ausgabe:
```
argv[0] = argument            //Programmname
argv[1] = Hallo
argv[2] = Welt
```

Starten Sie das Programm jetzt nochmals mit folgender Eingabe (»argument« sei wieder der Programmname):

```
if(strcmp(argv[j],"+") == 0)
```

20. Zeitroutinen <time.h>

20.1. Die Headerdatei <time.h> ▼

Es folgen einige Standardfunktionen der Headerdatei `<time.h>`, in denen Routinen für Zeit und Datum deklariert sind. Dazu ein kurzer Überblick über die speziellen (primitiven) Datentypen in dieser He

aderdatei und ihre Bedeutungen:

Tabelle 21.1 (Primitive) Datentypen und Struktur für Datum und Zeit

Typ	Bedeutung
size_t	arithmetischer Datentyp für Größenangaben
clock_t	arithmetischer Datentyp für CPU-Zeit
time_t	arithmetischer Datentyp für Datum- und Zeitangabe
struct tm	enthält alle zu einer Kalenderzeit (gregorianische) relevanten Komponenten

Laut ANSI C-Standard sollten in der Struktur `tm` folgende Komponenten enthalten sein:

Tabelle 21.2 Bedeutung der Strukturvariablen in struct tm

struct tm-Variable	Bedeutung
int tm_sec;	Sekunden (0–59)
int tm_min;	Minuten (0–59)
int tm_hour;	Stunden (0–23)
int tm_mday;	Monatstag (1–31)
int tm_mon;	Monate (0–11) (Januar = 0)
int tm_year;	ab 1900
int tm_wday;	Tag seit Sonntag (0–6) (Sonntag =0)
int tm_yday;	Tag seit 1. Januar (0–365) (1.Januar =0)
int tm_isdst;	Sommerzeit (tm_isdst > 0) Winterzeit (tm_istdst == 0) nicht verfügbar (tm_isdst < 0)

20.2. Datums – und Zeitfunktionen in <time.h> ▲

Die Zeit, mit welcher der Systemkern arbeitet, ist die Anzahl vergangener Sekunden, die seit dem **1. Januar 1970, 00:00:00 Uhr vergangen sind**. Diese Zeit wird immer mit dem

Datentyp **time_t** dargestellt und enthält das Datum und die Uhrzeit. Diese Zeit kann mit der Funktion

```
time_t time(time_t *zeitzeiger);
```

ermittelt werden. Wird für den Parameter zeitzeiger kein NULL-Zeiger verwendet, befindet sich an dieser Adresse die aktuelle Systemzeit. Hierzu ein kleines Listing, das die Zeit in Sekunden fortlaufend seit dem 1. Januar 1970 um 00:00:00 Uhr mithilfe der Funktion time() ausgibt:

```
/* time1.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef  __unix__
   #define clrscr() printf("\x1B[2J")
#else
   #include <stdlib.h>
   #define clrscr() system("cls")
#endif
int main(void) {
   time_t t;
   time(&t);
   while(1) {
      clrscr();
      printf("%ld\n",t);               // in Sekunden
      printf("Mit <STRG><C> bzw. <STRG><D> beenden!! ");
      time(&t);
   }
   return EXIT_SUCCESS;
}
```

Nach dem »Jahr 2000«-Problem steht zum Jahre 2038 das nächste Problem an, sollte bis dahin noch mit 32-Bit-Rechnern gearbeitet werden. Mittlerweile steht – beim Schreiben dieses Textes – die Sekundenzahl bei etwa einer Milliarde. time_t ist als long implementiert. Es gibt also Platz für etwa 2 Milliarden Sekunden. Sicherlich wird dieses Problem wieder im letzten Moment angegangen.

20.3. localtime() und gmtime() – Umwandeln von time_t in struct tm

Umwandlungs-Standardfunktionen:

```
struct tm *localtime(time_t *zeitzeiger);
struct tm *gmtime(time_t *zeitzeiger);
```

Beide Funktionen liefern als Rückgabewert die Adresse einer Zeitangabe vom Typ **struct tm.** Diese Struktur wurde bereits zu Beginn dieses Kapitels behandelt. Die Funktion localtime() wandelt die Kalenderzeit der Adresse time_t *zeitzeiger in lokale Ortszeit um – unter der Berücksichtigung von Sommer- und Winterzeit. **gmtime() dagegen wandelt die Kalenderzeit in die UTC-Zeit um.**

Hierzu ein Beispiel, welches die Eingabe eines Geburtsdatums erwartet und anschließend das Alter in Jahren, Monaten und Tagen ausgibt:

```
/* time2.c */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>
struct tm *tm_heute;
void today(void) {
   time_t t_aktuell;
   time(&t_aktuell);
   tm_heute = localtime(&t_aktuell);
   printf("Heute ist der ");
   printf("%d.%d.%d\n",
      tm_heute->tm_mday, tm_heute->tm_mon + 1, tm_heute->tm_year + 1900);
}
int main(void) {
   int tag, monat, jahr;
   unsigned int i=0, tmp;
   printf("Bitte gib Deinen Geburtstag ein!\n");
   printf("Tag : ");
   scanf("%d", &tag);
   printf("Monat : ");
   scanf("%d", &monat);
   printf("Jahr (jjjj) : ");
   scanf("%d", &jahr);
   today();
   if(tm_heute->tm_mon < monat) {
      i = 1;
      tmp=tm_heute->tm_mon+1-monat;
      monat=tmp+12;
   }
   else {
      tmp=tm_heute->tm_mon+1-monat;
      monat=tmp;
   }
   if(monat == 12) {
      monat = 0;
      i = 0;
   }
   printf("Sie sind %d Jahre %d Monate %d Tage alt\n",
      tm_heute->tm_year+1900-jahr-i,monat, tm_heute->tm_mday-tag);
   return EXIT_SUCCESS;
}
```

20.4. mktime() – Umwandeln von struct tm zu time_t

Jetzt zum Gegenstück der Funktionen `localtime()` und `gmtime()`:

```
time_t mktime(struct tm *zeitzeiger);
```

Auf diese Weise wird eine Zeit im `struct tm`-Format wieder umgewandelt in eine Zeit im `time_t`-Format. Ist die Kalenderzeit nicht darstellbar, gibt diese Funktion –1 zurück. Die echten Werte der Komponenten `tm_yday` und `tm_wday` in `zeitzeiger` werden ignoriert. Die ursprünglichen Werte der Felder, `tm_sec`, `tm_min`, `tm_hour`, `tm_mday` und `tm_mon`, sind nicht auf den durch die `tm`-Struktur festgelegten Bereich beschränkt. Befinden sich die Felder nicht im korrekten Bereich, werden diese angepasst.

Das heißt konkret: Wird z.B. das Datum 38.3.2001 eingegeben, muss die Funktion `mktime()` dieses Datum richtig setzen. Bei richtiger Rückgabe erhalten Sie entsprechende Werte für

`tm_yday` und `tm_wday`. Der zulässige Bereich für die Kalenderzeit liegt zwischen dem 1. Januar 1970 00:00:00 und dem 19. Januar 2038 03:14:07.

Ein Beispiel soll zeigen, wie Sie den genauen Wochentag durch diese Funktion ermitteln können:

```
/* time3.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
char *wday[] = {
   "Sonntag", "Montag", "Dienstag", "Mittwoch",
   "Donnerstag", "Freitag", "Samstag", "??????"
};
int main(void) {
   struct tm time_check;
   int year, month, day;
   /* Jahr, Monat und Tag eingeben zum
    * Herausfinden des Wochentags */
   printf("Jahr : ");
   scanf("%d", &year);
   printf("Monat: ");
   scanf("%d", &month);
   printf("Tag  : ");
   scanf("%d", &day);
   /* Wir füllen unsere Struktur struct tm time_check
    * mit Werten */
   time_check.tm_year = year - 1900;
   time_check.tm_mon = month - 1;
   time_check.tm_mday = day;
   /* 00:00:01 Uhr */
   time_check.tm_hour  = 0;
   time_check.tm_min   = 0;
   time_check.tm_sec   = 1;
   time_check.tm_isdst = -1;
   if(mktime(&time_check) == -1)
      time_check.tm_wday = 7;  /* = Unbekannter Tag */
   /* Der Tag des Datums wird ausgegeben */
   printf("Dieser Tag ist/war ein %s\n",
      wday[time_check.tm_wday]);
   return EXIT_SUCCESS;
}
```

20.5. asctime() und ctime() – Umwandeln von Zeitformaten in einen String

Mit zwei Funktionen können die beiden Zeitformen `struct tm` und `time_t` in einen String konvertiert werden. Hier die Syntax der beiden:

```
char *asctime(struct tm *zeitzeiger);
char *ctime(time_t *zeitzeiger);
```

difftime() – Differenz zweier Zeiten

Wird eine Differenz zwischen zwei Zeiten benötigt, lässt sich dies mit der folgenden Funktion ermitteln:

```
double difftime(time_t zeit1, time_t zeit0);
```

Diese Funktion liefert die Differenz von `zeit1` minus `zeit0` als `double`-Wert zurück. Hierzu ein einfaches und kurzes Beispiel:

```
/* time5.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void) {
   time_t start, stop;
   double diff;
   printf("Einen Augenblick bitte ...\n");
   start=time(NULL);
   while((diff=difftime(stop=time(NULL),start)) != 5);
   printf("%.1f sek. vorbei!!\n",diff);
   return EXIT_SUCCESS;
}
```

Das Programm wartet fünf Sekunden, bis es einen entsprechenden Text ausgibt. Bei

```
while((diff=difftime(stop=time(NULL),start)) !=5);
```

wurde die Funktion `time()` gleich in der Funktion `difftime()` ausgeführt. Natürlich ist dies nicht so gut lesbar, aber es erfüllt denselben Zweck wie:

```
while((diff=difftime(stop,start)) != 5)
   stop=time(NULL);
```

20.6. clock() – Verbrauchte CPU-Zeit für ein Programm

Eine weitere häufig gestellte Frage lautet: Wie kann ich herausfinden, wie lange das Programm schon läuft? Sie können dies mit folgender Funktion herausfinden:

```
clock_t clock();
```

Diese Funktion liefert die verbrauchte CPU-Zeit seit dem Programmstart zurück. Falls die CPU-Zeit nicht verfügbar ist, gibt diese Funktion –1 zurück. Wenn Sie die CPU-Zeit in Sekunden benötigen, muss der Rückgabewert dieser Funktion durch `CLOCKS_PER_SEC` dividiert werden. Beispiel:

```
/* runtime.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void) {
   clock_t prgstart, prgende;
   int c;
   prgstart=clock();
   printf("Geben Sie etwas ein und beenden Sie mit #\n");
   printf("\n > ");
   while((c=getchar())!= '#')
      putchar(c);
   prgende=clock();
   printf("Die Programmlaufzeit betrug %.2f Sekunden\n",
      (float)(prgende-prgstart) / CLOCKS_PER_SEC);
   return EXIT_SUCCESS;
}
```

20.7. 11.20 Rekursive Funktionen ▼

Kurz gesagt ist eine Rekursion eine Funktion, die sich selbst aufruft und sich selbst immer wieder neu definiert. Damit sich aber eine Rekursion nicht unendlich oft selbst aufruft, sondern irgendwann auch zu einem Ergebnis kommt, benötigen Sie unbedingt eine so genannte Abbruchbedingung. Sonst kann es irgendwann passieren, dass Ihr Computer abstürzt, da eine Funktion, die sich immer wieder selbst aufruft, eine Rücksprungadresse, den Wert der Variablen und – falls noch nicht freigegeben – den Rückgabewert speichert. Der dafür zur Verfügung stehende Speicher (Stack) wird so aber unweigerlich irgendwann voll sein beziehungsweise überlaufen (Stacküberlauf oder Stack Overflow).

20.7.1. 11.20.1 Exkurs: Stack ▼▲

Der Stack wurde bereits öfters erwähnt. Er soll deshalb im Folgenden näher betrachtet werden.

Der Stack dient dazu, den Speicherbereich für Funktionsaufrufe zu verwalten. Dieser Speicherbereich ist dynamisch, was bedeutet, dass der Speicher bei Bedarf automatisch anwächst und wieder schrumpft. Der Compiler, der diesen Stack verwaltet, legt hier alle Daten ab, die er zur Verwaltung von Funktionsaufrufen benötigt.

Wenn eine Funktion aufgerufen wird, erweitert der Compiler den Stack um einen Datenblock. In diesem Datenblock werden die Parameter, die lokalen Variablen und die Rücksprungadresse zur aufrufenden Funktion angelegt. Dieser Datenblock wird als Stack-Frame oder Stackrahmen bezeichnet.

Der Datenblock bleibt so lange bestehen, bis diese Funktion wieder endet. Wird in ihm aber eine weitere Funktion aufgerufen, wird ein weiterer Datenblock auf den (richtig wäre: unter den) aktuellen gepackt. Der Stack wächst nach unten an. Am Anfang des Stacks befindet sich der Startup-Code, der die main()-Funktion aufruft, welche eine Position unter dem Startup-Code liegt. An unterster Stelle befindet sich immer die aktuelle Funktion, die gerade ausgeführt wird. Eine Position – oder besser: einen Datenblock – darüber liegt die aufrufende Funktion in der Wartestellung. Sie wartet auf die Beendigung der nächsten aufgerufenen Funktion. Mit diesem Wissen über den Stack können Sie sich wieder den Rekursionen widmen.

20.7.2. 11.20.2 Rekursionen und der Stack ▼▲

Mit Rekursionen haben Sie die Möglichkeit, den Computer zu etwas zu bewegen, was ihn intelligenter erscheinen lässt. Ein Beispiel wäre etwa Schach. Wenn Sie einen Zug machen, gehen Sie zuerst alle Möglichkeiten durch, um den Gegner in Bedrängnis bzw. den gegnerischen König in Gefahr zu bringen oder gar schachmatt zu setzen. Das ist eine logische Denkweise des Menschen. Mit einer Rekursion ist es ebenfalls möglich, den Computer eine Situation sooft durchgehen zu lassen, bis er auf eine Lösung kommt oder auch nicht. Man spricht dabei vom »Trial and Error-Verfahren« (Versuch und Irrtum). Ein Beispiel: Sie

bedrohen den König des Computers. Der Computer geht dann alle Züge durch, um den König aus dieser Bedrohung zu befreien, und dann, in einem zweiten Schritt, geht er nochmals alle Züge durch, die Sie als Nächstes theoretisch machen könnten. Je nachdem, wie tief die Rekursion gehen soll. Zum besseren Verständnis folgt ein konkretes Beispiel.

Eine Funktion soll zwei Zahlen dividieren. Der ganzzahlige Rest der Division soll angegeben werden. Zum Beispiel: 10/2=5 oder 10/3=3 Rest 1. Das Programm darf aber nicht die Operatoren / und % verwenden. Die Lösung soll die Form einer rekursiven Funktion haben:

```
int divide(int x, int y) {
   if(x >= y)
      return (1 + divide(x - y, y));
   if(x)
      printf("Zahl nicht teilbar -> Rest: %d -> ", x);
   return 0;
}
```

Hier ein Fall, in dem der Funktion beispielsweise die Werte x=8 und y=2 übergeben werden:

```
/* Funktionsaufruf */
printf("8/2 = Ergebnis : %d\n", divide(8, 2));
```

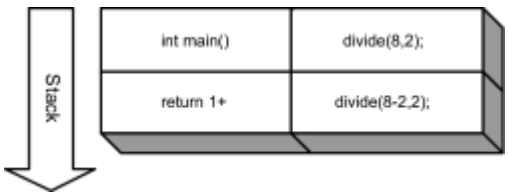
Innerhalb der Funktion wird zunächst die Abbruchbedingung überprüft:

```
if(x >= y)
```

Da die Bedingung für x=8 und y=2 wahr ist, wird die nächste Anweisung ausgeführt:

```
return 1 + divide(x - y, y);
```

Die Funktion gibt mittels return die Summe 1+divide(x-y,x) zurück. Damit wird, bevor das Ergebnis endgültig zurückgegeben wird, die Funktion divide erneut aufgerufen. Die Funktion ruft sich also selbst auf. Hiermit beginnt die Rekursion. Aber was passiert jetzt mit dem Rückgabewert 1? Sehen Sie sich das Beispiel zum besseren Verständnis in der Abbildung an:



Hier klicken, um das Bild zu Vergrößern

Abbildung 11.5 Erster Rekursiver Aufruf

Auf den Stack wurde zuerst die main()-Funktion gelegt, da diese zuerst die Funktion divide() aufgerufen hat. Hier ist quasi gespeichert, wie Ihr Programm wieder zur main()-Funktion zurückkommt. Sie können sich das etwa so vorstellen: Bei jedem Funktionsaufruf in einem Programm, unabhängig davon, ob rekursiv oder nicht, wird der aktuelle Zustand der main()-Funktion eingefroren und auf dem Stack abgelegt. Damit das Programm weiß, wo die Adresse der main()-Funktion ist, wird auf dem Stack eine Rücksprungadresse mit abgelegt. Zurück zur Programmausführung des konkreten Beispiels. Die Funktion hat sich also selbst aufgerufen mit der Anweisung:

```
return 1 + divide(x - y, y);
```

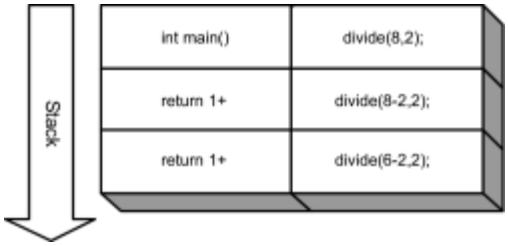
In Zahlen also divide(8–2,2) mit den Werten x=8 und y=2. Im abermaligen Funktionsaufruf wird erneut überprüft:

```
if(x >= y)
```

Da x=6 und y=2 und somit die if-Abfrage wieder wahr ist, geht die Programmausführung wieder in der nächsten Zeile weiter. Es folgt ein erneuter Selbstaufruf der Funktion divide():

```
return 1 + divide(x - y, y);
```

Also wird Folgendes auf dem Stack abgelegt (siehe Abbildung).



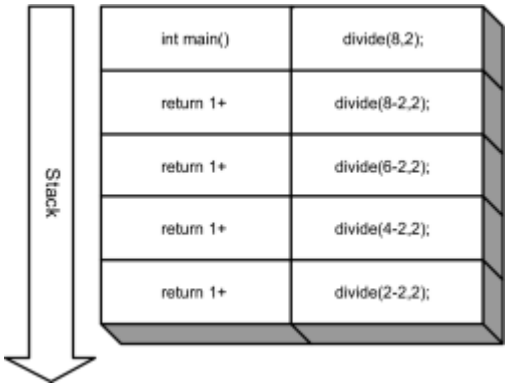
Hier klicken, um das Bild zu Vergrößern

Abbildung 11.6 Zweiter Rekursiver Aufruf

Nun liegt auf dem Stack zweimal der Rückgabewert 1, inklusive der Rücksprungadressen (diese sind hier nicht mit abgebildet). Jetzt wiederholt sich das ganze Spiel noch zweimal, bis es auf dem Stack folgendermaßen aussieht (siehe Abbildung 11.7).

Der Funktionswert für x im Aufruf der Funktion ist mittlerweile auf 2 reduziert worden. Danach wird erneut die Funktion divide() aufgerufen und zwar mit den Werten:

```
divide(2-2,2)
```



Hier klicken, um das Bild zu Vergrößern

Abbildung 11.7 Stack nach vier rekursiven Aufrufen

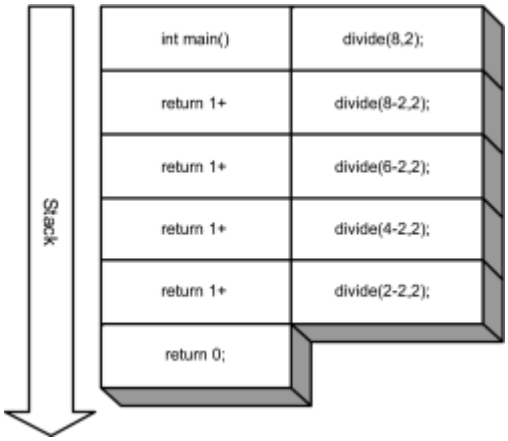
Jetzt wird die Abbruchbedingung aktiv:

```
if(x >= y)
```

Denn jetzt ist x=0 und y=2 und somit wird die Programmausführung nicht mehr in der nächsten Zeile ausgeführt. Die nächste Abfrage

```
if(x)
```

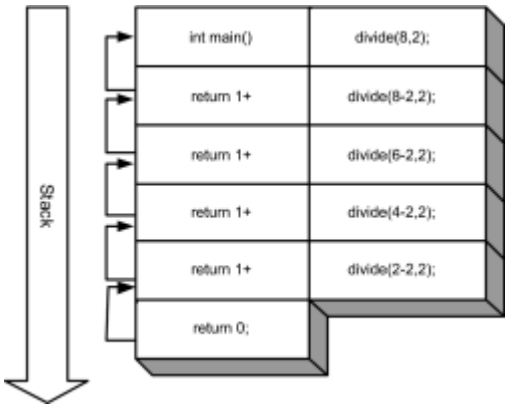
dient dazu, den Rest auszugeben, falls x ungleich 0 sein sollte. In unserem Beispiel gibt es keinen Rest. Es wird also der Wert 0 (return 0) zurückgegeben. Das Programm muss nun zur nächsten Rücksprungadresse gehen, da sich die Funktion ja beendet hat. Nochmals ein Blick zum Stack:



Hier klicken, um das Bild zu Vergrößern

Abbildung 11.8 Die Abbruchbedingung greift jetzt ein

Der Rückgabewert 0 wurde von dem Funktionsaufruf divide(2–2,2) erzeugt. Dorthin führt auch die Rücksprungadresse, also return 0+1. Die nächste Rücksprungadresse wurde von divide(4–2,2) erzeugt, also folgt return 0+1+1; anschließend return 0+1+1+1 und zuletzt return 0+1+1+1+1. Die main-Funktion bekommt dann den Rückgabewert 0+1+1+1+1, also 4, und das ist auch korrekt, denn 8/2 ist 4.



Hier klicken, um das Bild zu Vergrößern

Abbildung 11.9 Addieren der einzelnen Rückgabewerte auf dem Stack

Sie werden sich möglicherweise fragen, welche Vorteile ein solches Programm gegenüber einem Programm der folgenden Form hat:

```
/* divide.c */
#include <stdio.h>
```

```
#include <stdlib.h>
int main(void) {
   int x = 8, y = 2;
   printf("%d ", x/y);
   if(x % y)
      printf("Rest = %d\n",x%y);
   return EXIT_SUCCESS;
}
```

Dieses Programm erfüllt doch denselben Zweck und ist einfacher! Sie haben Recht, das rekursive Programm ist zum einen schwieriger und zum anderen langsamer, da ständig etwas auf dem Stack gepusht und wieder gepopt werden muss.

Kurz gesagt: Die rekursive Lösung ist die schlechtere in diesem Beispiel. Schlimmer noch, die rekursive Lösung verbraucht viel Speicherplatz zum Anlegen von Parametern, lokalen Variablen, Rückgabewerten und Rücksprungadressen. Ein Beispiel: Sie wollen die Zahl 1.000.000 durch 2 teilen. Für die zwei Parameter x und y benötigen Sie schon acht Byte pro Aufruf. Für den Rückgabewert (return 1) werden weitere vier Bytes benötigt, genauso wie für die Rücksprungadresse. Das heißt, Sie verwenden für eine Ablage auf dem Stack 16 Byte. Wenn Sie die Zahl 1.000.000 durch 2 teilen, bedeutet dies, dass auf dem Stack 500.000 Werte zu je 16 Bytes liegen. Das sind ca.7,6 Megabytes Arbeitsspeicher, die Sie durch eine rekursive Lösung eines solch einfachen Problems verschwenden.

Warum also Rekursionen anwenden, wenn die direkte Lösung oftmals die bessere ist? In späteren Programmen werden Sie einige Beispiele kennen lernen (so genannte binäre Bäume), die ohne Rekursion nicht so einfach realisierbar wären.

Die Rekursion will ich Ihnen anhand von einigen Beispielen noch näher erläutern. Die verwendeten Programme sollen nur die Rekursion verdeutlichen. Es ist einleuchtend, dass die Programme ansonsten auch einfacher und meistens besser lösbar sind. Es sind typische, klassische Beispiele.

20.7.3. 11.20.3 Fakultät ▼▲

Es soll eine Funktion geschrieben werden zum Berechnen der Fakultät der Zahl n. Die Fakultät der Zahl 6 ist zum Beispiel: 1*2*3*4*5*6=720. Die Fakultät von 10 ist 1*2*3*4*5*6*7*8*9*10=3.628.800.

Wie schreiben Sie die Funktion am besten? Zuerst benötigen Sie eine Abbruchbedingung. Es muss lediglich überprüft werden, ob die Zahl, von der Sie die Fakultät berechnen wollen, ungleich 0 ist:

```
/* fakul.c */
#include <stdio.h>
#include <stdlib.h>
long fakul(long n) {
   if(n)
      return n * fakul(n-1);
   return 1;
}
int main(void) {
```

```
   printf("Fakultät von 5 = %ld\n",fakul(5));
   printf("Fakultät von 9 = %ld\n",fakul(9));
   return EXIT_SUCCESS;
}
```

Die Funktion rechnet so lange n*n-1, bis n den Wert 0 hat. Denn n*0 würde sonst das Ergebnis 0 ergeben. Bei fakul(5) wären dies dann 5*4*3*2*1=120, wobei n*1 eigentlich auch eingespart werden kann, denn mit n*1 wird sich der Wert nicht ändern. Natürlich will ich Ihnen die alternative direkte Lösung des Problems nicht vorenthalten:

```
long fakul(int n) {
   int x = n;
   while(--x)
      n *= x;
   return n;
}
```

20.7.4. 11.20.4 Fibonacci-Zahlen ▼▲

Die Fibonacci-Zahlen sollen rekursiv berechnet werden. Fibonacci-Zahlen sind z.B. 1, 2, 3, 5, 8, 13, 21, ...

Errechnet werden können sie mittels ... 1+2=3, 2+3=5, 3+5=8, 5+8=13. Nach Formel also:

F(n+2)=F(n+1) +F(n)

Hierzu der Code:

```
/* fibo.c */
#include <stdio.h>
#include <stdlib.h>
long fibo(long n) {
   if(n)
      return (n <= 2) ? n : fibo(n-2) + fibo(n-1);
}
int main(void) {
   long f;
   long i=0;
   printf("Wie viele Fibonacci-Zahlen wollen Sie ausgeben:");
   scanf("%ld",&f);
   while(i++ < f)
      printf("F(%ld) = %ld\n", i, fibo(i));
   return EXIT_SUCCESS;
}
```

20.7.5. 11.20.5 Größter gemeinsamer Teiler (GGT) ▲

Es folgt ein Listing zum Ermitteln des größten gemeinsamen Teilers zweier Zahlen. Natürlich wird dafür der rekursive Weg eingeschlagen. Auch hier muss zuerst eine Abbruchbedingung gefunden werden. Sie haben drei Möglichkeiten zum Errechnen des GGT zweier Zahlen:

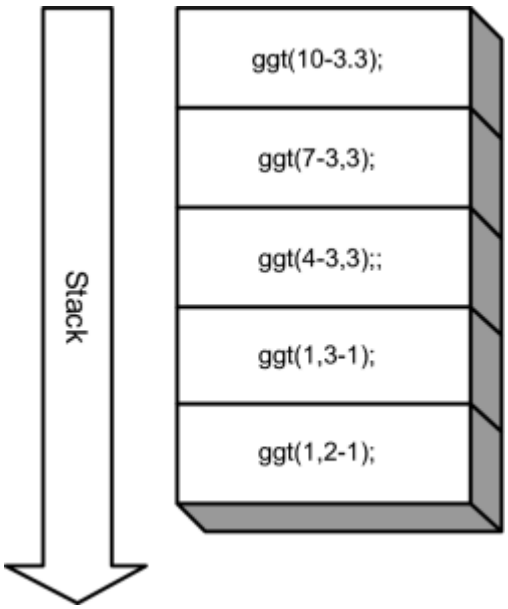
```
ist Zahl1 == Zahl2 dann Ergebnis = Zahl1
```

```
ist Zahl1  > Zahl2 dann Ergebnis = ggT(Zahl1-Zahl2, Zahl2)
ist Zahl1  < Zahl2 dann Ergebnis = ggT(Zahl1, Zahl2-Zahl1)
```

Das Programm sieht folgendermaßen aus:

```
/* ggt1.c */
#include <stdio.h>
#include <stdlib.h>
unsigned long ggt(unsigned long a, unsigned long b) {
   if(a==b)
      return a;
   else if(a < b)
      return ggt(a, b-a);
   else
      return ggt(a-b, b);
}
int main(void) {
   unsigned long a, b;
   printf("ggt = größter gemeinsamer Teiler\n");
   printf("Zahl 1: ");
   scanf("%lu",&a);
   printf("Zahl 2: ");
   scanf("%lu",&b);
   printf("Der ggT von %lu und %lu ist %lu\n", a, b, ggt(a,b));
   return EXIT_SUCCESS;
}
```

Beispiel: Sie geben für a=10 und b=3 ein. Folgende Wertepaare werden auf den Stack gelegt, bis das Programm den GGT von 1 zurückgibt:



Hier klicken, um das Bild zu Vergrößern

Abbildung 11.10 Rekursive Ermittlung des größten gemeinsamen Teilers

Eine alternative direkte Lösung wäre gewesen:

```
/* ggt2.c */
#include <stdio.h>
#include <stdlib.h>
unsigned long ggt(unsigned long a, unsigned long b) {
```

```
   unsigned long count;
   if(a==b)
      return a;
   else if( (a % b) == 0)
      return b;
   else
      for(count = b; count > 0; count--) {
         if( ( (a % count) + (b % count) ) == 0)
            return count;
      }
}
int main(void) {
   unsigned long a, b, c;
   printf("ggt = größter gemeinsamer Teiler\n");
   printf("Zahl 1: ");
   scanf("%lu",&a);
   printf("Zahl 2: ");
   scanf("%lu",&b);
   if(a<b) { /* a und b vertauschen */
      c=a; a=b; b=c;
   }
   printf("Der ggT von %lu und %lu ist %lu\n", a, b, ggt(a,b));
   return EXIT_SUCCESS;
}
```

Nun soll der größte gemeinsame Teiler von beliebig vielen Zahlen ermittelt werden. Die Schwierigkeit liegt bei diesem Beispiel aber nicht in der rekursiven Funktion, sondern in der main()-Funktion. Sie könnten die Funktion GGT, wie diese eben geschrieben wurde, benutzen, ohne sie zu verändern. Zuvor möchte ich Ihnen aber noch eine zweite Möglichkeit demonstrieren, wie Sie den GGT ermitteln können. Hier die Funktion:

```
unsigned long ggt(unsigned long a, unsigned long b) {
   if(b==0)
      return a;
   return ggt(b, a % b);
}
```

Jetzt lassen sich womöglich die Vorteile einer Rekursion erkennen. Die rekursive Funktion erfüllt den gleichen Zweck wie die beiden Funktionen GGT zuvor. Mit return ggt(b, a%b) rufen Sie die Funktion erneut auf. Wenn a%b==0 ergibt, haben Sie ja den GGT durch b an a übergeben. Hier die main()-Funktion zum Ermitteln des GGT mehrerer Zahlen:

```
/* ggt3.c */
#include <stdio.h>
#include <stdlib.h>
unsigned long ggt(unsigned long a, unsigned long b) {
   if(b == 0)
      return a;
   return ggt(b, a % b);
}
int main(void) {
   unsigned long a, b;
   printf("ggt = größter gemeinsamer Teiler(mit 0 beenden)\n");
   printf("Zahl> ");
   scanf("%lu", &a);
   printf("Zahl> ");
   scanf("%lu", &b);
   a=ggt(a, b);
   while(1) {
      printf("Zahl> ");
```

```
      scanf("%lu", &b);
      if(b==0)
         break;
      a=ggt(a, b);
   }
   printf("-------->ggt = %lu\n", a);
   return EXIT_SUCCESS;
}
```

An dem Programm wurde nicht viel verändert. Es kam lediglich die while-Schleife hinzu, die Sie mit der Eingabe 0 beenden können.

Wichtig ist, dass Sie bei jedem Schleifendurchlauf den größten gemeinsamen Teiler an a und die neue Zahl an b übergeben. Somit wird immer der GGT aller Zahlen aktualisiert.

Als letztes Beispiel will ich Ihnen zeigen wie Sie eine rekursive Funktion zum Umwandeln von Dezimalzahlen nach Dualzahlen verwenden können. Um bspw. aus der Zahl 10 die entsprechende Dualzahl 1010 zu machen, ist folgender Vorgang nötig:

```
-> Solange die Zahl ungleich Null ->
-> Zahl % 2 = kein Rest dann 0 oder = Rest dann 1 ->
-> Zahl = Zahl / 2
```

Auf die Zahl 10 angewendet sieht dieser Vorgang wie folgt aus:

```
10/2 = 5 kein Rest -> 0
5/2  = 2 Rest 1    -> 1
2/2  = 1 kein Rest -> 0
1/2  = 0 Rest 1    -> 1
```

Damit liegen auf dem Stack (umgekehrte Reihenfolge):

```
1
0
1
0
```

Hier das Beispiel dazu:

```
/* dez2bin.c */
#include <stdio.h>
#include <stdlib.h>
#define ulong unsigned long
void dez2bin(ulong dez) {
   if(dez) {
      dez2bin(dez / 2);
      printf("%lu", dez % 2);
   }
}
int main(void) {
   ulong dezimal;
   printf("Dezimalzahl in Dualzahl konvertieren\n");
   printf("Welche Zahl : ");
   scanf("%lu",&dezimal);
   printf("Dezimal = %lu Dual = ",dezimal);
   dez2bin(dezimal);
   printf("\n");
   return EXIT_SUCCESS;
```

```
}
```

Dies genügt nun zum Thema Funktionen. In einem späteren Kapitel wird es wieder aufgegriffen, wenn es darum geht, Funktionen mit beliebig vielen Parametern zu erstellen. Dafür müssen jedoch zuerst die Zeiger besprochen werden.