1. Die Verwendung von Zeigern und Variablen in C

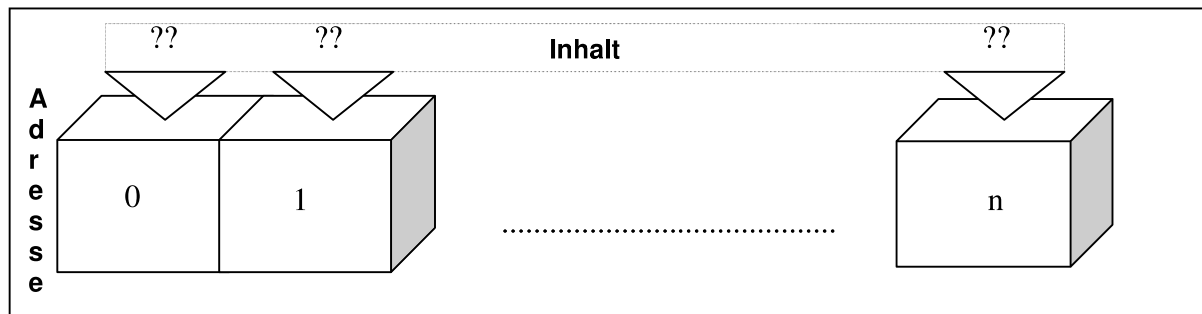
Um dieses Thema einzuleiten ist zunächst zu sagen, dass Zeiger in C und auch in anderen Programmiersprachen vor allem Anfänger erhebliche Probleme bereiten, aber auch für Profis nicht immer einleuchtend sind. Deshalb werden gerade in diesem Bereich sehr viele Fehler gemacht.

Um aber Zeiger und Variablen verstehen zu lernen, genügt es nicht, sich einfach mit den Begrifflichkeiten auseinander zusetzen, sondern man sollte sich als erstes den Aufbau und den Zweck eines Speichers verdeutlichen. Die folgenden Betrachtungen dienen nur zur Veranschaulichung, sind aber auch in der Praxis in etwa so umgesetzt.
Generell kann man den Speicher als eine Ansammlung von Postfächern betrachten, die ohne Lücken durchnumeriert und deren Nummer die sog. **Adresse** ist (ähnlich der Hausnummer eines Hauses in einer Strasse). Jedem einzelnen „Postfach“ wird hierbei der Begriff **Speicherstelle** zugeordnet. Somit kann jede Speicherstelle eindeutig angesprochen (adressiert) werden. Man verwendet hierzu einfach die der Speicherstelle zugeordnete Nummer. Die Anzahl der Adressen richtet sich danach, wie viel Speicherbausteine sich in dem jeweiligen Rechner befinden.

Adresse
0
1
. . .
n

Abbildung 1 - Aufbau eines Speichers

Nun hat ein Postfach (Speicherstelle) nicht nur eine Adresse, sondern auch einen Inhalt. Um dies zu verdeutlichen stellt man nun das Postfach dreidimensional dar, um den Inhalt der Speicherstelle (des Postfachs) zu erblicken:



Dabei ist der Inhalt einer Speicherstelle immer eine Ganzzahl (und wird erst später interpretiert, was die Zahl in Wirklichkeit darstellen soll, z. B. Buchstabe, Zeiger, Fließkommazahl, usw.). Generell ist zu Beginn der Inhalt jeder Speicherstelle undefiniert (hier mit ?? verdeutlicht) und muss vom Programmierer zunächst initialisiert (auf einen definierten Wert gesetzt) werden. Das heißt, wenn man eine Variable verwendet ist deren Inhalt zunächst irgendein Wert (siehe weiter unten).

1.1 Variablen

Möchte man nun einen Wert in einer Speicherstelle ablegen - was in vielen Programmen die Hauptaufgabe darstellt – und damit arbeiten (addieren, subtrahieren, usw.) so muss man eine Variable deklarieren. Dies geschieht in C nach folgender Notation:

```
Datentyp Variablenname;
```

Datentyp ist einer der in C definierten Datentypen **int, float, double, char**. Wobei es optional auch noch möglich ist den Variablen gleich zu Beginn einen Wert zuzuweisen, da wie oben schon gesagt, der Inhalt der Variablen undefiniert ist. Deshalb sollte generell eine Variablendeklaration z. B. wie folgt aussehen:

```
int     a = 10;
float   b = 10.0;
char    c = 'A';
int     d;           /* Nicht initialisierte Variable (schlecht) */
```

Wie sieht nun oben genanntes Beispiel im Speicher aus? Folgendes Beispiel ist völlig willkürlich und wird bei jedem Programmstart und bei jedem Computer anders aussehen:

Adresse	Inhalt
. . .	
4	10
. .	
100	10.0
101	,A'
. . .	
303	?? (Nicht bekannt, da nicht initialisiert)

Wie ich oben schon erwähnt habe, befinden sich im Speicher nur ganzzahlige Werte die erst später vom Programm interpretiert werden. Zur Veranschaulichung habe ich aber die Werte mit den entsprechenden Datentypen so angezeigt, wie sie oben im Beispiel verwendet wurden. Es ist anzumerken, dass der Programmierer keinen Einfluss darauf hat, wo und auf welche Art und Weise die Variable im Speicher abgelegt wird (im Gegensatz zu Zeigern, siehe weiter unten). Dies erledigt der Compiler für den Programmierer. Um dennoch zu ermitteln, an welcher Speicherstelle (Adresse) die Variable abgelegt wurde, benutzt man den **Adressoperator &**[1]:

```
#include <stdio.h>

main()
{
  int   a = 10;
  float b = 10.0;
  char  c = 'A';
  int   d;              /* Nicht initialisierte Variable (schlecht) */

  printf („%d\n", a);    /* gibt den Wert der Variablen a aus */
  printf („%x\n\n", &a); /* gibt die Adresse der Variablen a aus */

  printf („%f\n", b);    /* gibt den Wert der Variablen b aus */
  printf („%x\n\n", &b); /* gibt die Adresse der Variablen b aus */

  printf („%c\n", c);    /* gibt den Wert der Variablen c aus */
  printf („%x\n\n", &c); /* gibt die Adresse der Variablen c aus */

  printf („%d\n", d);    /* gibt den Wert der Variablen d aus (hier undefiniert) */
  printf („%x\n\n", &d); /* gibt die Adresse der Variablen d aus */
}
```

Dieses Beispielprogramm angewandt auf die obig dargestellte Speicherzusammensetzung würde nun folgende Ausgabe erzeugen:

```
10
0x4

10.0
0x100
```

[1] Adressen werden normalerweise in hexadezimaler Schreibweise ausgegeben. Deshalb kann man beim printf den Formatierer **%x** und **%X** angeben, der die Zahlenwerte in hexadezimale Werte umwandelt und dabei die C Notation **0x** vor jede Hexadezimalzahl voranstellt.

```
A
0x101

324
0x303
```

Vergleicht man die Ausgabe mit dem Beispielprogramm, so scheint alles korrekt zu sein, mit Ausnahme der Variablen d. Hier wird ein Wert ausgegeben, der niemals im Programmcode erscheint. Dies ist vollkommen richtig, da die Variable nicht initialisiert wurde, befindet sich ein undefinierter Wert in der Speicherstelle. D. h. es wird der Wert ausgegeben, der sich aktuell in der Speicherstelle befindet (hier: 324 als Beispiel).

1.2 Zeiger

Zeiger sind in C nicht trivial und werden sehr ungern verwendet (aufgrund später folgender Gesichtspunkte). Zu Beginn sollte man aber zunächst einmal klären was eigentlich ein Zeiger ist:

Ein Zeiger ist eine Variable, die auf eine Speicherstelle zeigt.

Mit anderen Worten: Ein Zeiger ist eine normale Variable und unterscheidet sich von den normalen Variablen nur dadurch, dass der Inhalt der Variablen keinen Wert darstellt (mit dem gerechnet wird), sondern eine Adresse, die auf eine Speicherstelle (also auf eine andere Variable) verweist. Sehen wir uns zunächst an, wie man einen Zeiger deklariert:

```
Datentyp *Variablenname;
```

Auf den ersten Blick wird ein Zeiger genau so wie eine normale Variable deklariert, doch schreibt man unmittelbar vor den Variablennamen einen **Stern ***. Auch hier gilt, dass der Inhalt des Zeigers undefiniert ist und erst initialisiert werden muss. Beispiel:

```
int *a;        /* Nicht initialisiert (schlecht) */
int *b = 10;
```

Wie sieht aber ein Zeiger im Speicher aus? Nun, genauso wie normalen Variablen:

Adresse	Inhalt
0 (Hier ist die Adresse des Zeigers a)	?? (Inhalt von a, aber nicht initialisiert, daher undefiniert!)
. . .	
15 (Hier ist die Adresse des Zeigers b)	10 (Hier befindet sich der Inhalt des Zeigers b, Inhalt ist definiert, da mit 10 initialisiert).
. . .	
n	

Auch bei Zeigern hat der Programmierer keinen Einfluss an welcher Speicherstelle der Inhalt des Zeigers abgelegt wird, wohl aber auf den Inhalt (mit einer speziellen Notation!). Beginnen wir mit den Operationen, die uns schon von den Variablen bekannt sind. Dies soll ein kleines Beispiel verdeutlichen:

```
#include <stdio.h>

main()
{
  int *a;       /* Hier wird der Zeiger a deklariert (aber undefiniert)*/
  int *b = 10;  /* Hier wird der Zeiger b deklariert (initialisiert mit dem Wert 10) */

  /* Hier geben wir wieder Inhalt und Adresse der Zeiger aus */
```

```
  printf („%x\n", a);    /* gibt den Wert des Zeigers a aus */
  printf („%x\n\n", &a); /* gibt die Adresse der Zeigers a aus */

  printf („%x\n", b);    /* gibt den Wert des Zeigers b aus */
  printf („%x\n\n", &b); /* gibt die Adresse des Zeigers b aus */
}
```

Mit obigem Beispiel und obiger Adressentabelle sieht die Bildschirmausgabe wie folgt aus:

```
0x345
0x0

0x10
0x15
```

Bei diesem Beispiel ist der Inhalt von a, wie schon oben erwähnt, nicht definiert und wurde bei der Bildschirmausgabe auch mit einem willkürlichen Zahlenwert ausgegeben (hier: 345). Vergleicht man die Ergebnisse der Zeigerausgabe mit der Ausgabe der Variablen, so stellt man nur einen Unterschied bei den Adressen und Inhalten fest. Bis hierher verhalten sich also Variablen und Zeiger gleich, wenngleich man Zeigerinhalte nicht zum Speichern von Zahlenwerten verwenden sollte, sondern nur zum Speichern von Adressen.

Kommen wir nun zu einer dritten Operation die den Zeiger von den Variablen unterscheidet. Wie oben schon erwähnt ist ein Zeiger eine Variable, die auf eine Speicherstelle, also auf eine Variable zeigt. Wie ist das zu verstehen? Deklarieren wir uns hierzu eine normale Variable und einen Zeiger:

```
int a;
int *b;
```

Im Speicher werden die Variable und der Zeiger wir folgt abgelegt:

Adresse	**Inhalt**
0 (Hier ist die Adresse der Variablen a)	?? (Inhalt von a, aber nicht initialisiert, daher undefiniert!)
. . .	
15 (Hier ist die Adresse des Zeigers b)	?? (Hier befindet sich der Inhalt des Zeigers b, Inhalt ist undefiniert)
. . .	
n	

weist man dem Zeiger b nun die Adresse der Variablen a (hier: 0) zu

```
b = &a;
```

so sieht der Speicher folgendermaßen aus:

Adresse	**Inhalt**
0 (Hier ist die Adresse der Variablen a)	?? (Inhalt von a, aber nicht initialisiert, daher undefiniert!)
. . .	
15 (Hier ist die Adresse des Zeigers b)	0 (Adresse von a wurde b zugewiesen)
. . .	
n	

Dies bedeutet: b beinhaltet nun die Adresse von a. Man sagt: „b zeigt auf a“. Dies kann man folgendermaßen verdeutlichen:

Adresse
0 (=a)
. . .
15 (=b)
. . .
n

Das heißt aber, wenn b auf a zeigt, so sollte es eigentlich möglich sein über b den Inhalt von a zu ermitteln. Das geht so:

```
#include <stdio.h>

main()
{
  int a = 10;    /* Nun wird a mit einem Wert (10) initialisiert */
  int *b;

  b = &a;        /* b wird die Adresse von a zugewiesen */

  /* Ausgabe */
  printf („%d\n“, a);      /* Ausgabe des Inhaltes von a */
  printf („%x\n\n“,&a);    /* Ausgabe der Adresse von a */

  printf („%x\n“, b);      /* Ausgabe des Inhaltes von b */
  printf („%x\n“, &b);     /* Ausgabe der Adresse von b */
  printf („%d\n\n“,*b);    /* Ausgabe des Inhaltes von a */
}
```

Mit obiger Speichertabelle und obigem Beispielprogramm sieht die Ausgabe dann wie folgt aus:

```
10
0x0

0x0
0x15
10
```

Man sieht also, dass mit der Verwendung des sog. **Dereferenzierungsoperators ***, der vor die Variable b gestellt wird, der Inhalt an der Speicherstelle ermittelt wird, der sich aktuell an der Speicherstelle der Adresse b befindet. Dabei bedeutet dereferenzieren nur, dass sich in der Speicherstelle b eine Referenz (ein Verweis) auf eine andere Speicherstelle befindet und diese durch den * Operator aufgelöst wird, also so als würde man direkt auf die Speicherstelle zugreifen.

1.3 Zeiger und Funktionsparameter

Auch bei Funktionen treten mehrfach Probleme mit Zeigern auf. Dabei soll hier aber nur auf praktikumsrelevante Probleme eingegangen werden. Jeder hat in seinem Praktikum schon einige Funktionen verwendet (z. B. printf, scanf). Hier kommt es vor, dass man manchmal eine Variable ohne den **Adressoperator &** verwenden muss und manchmal mit. Warum? Sehen wir uns zunächst einige Funktionsdefinitonen an. Zwei Beispiele sollen zunächst genügen.

```
void constint (const int a)
{
  printf („%d“, a);
}

void normalint (int a)
{
  printf („%d“, a);
```

```
}
```

Generell ist bei Funktionen zu sagen, dass für jeden Parameter eine lokale Kopie angelegt wird, d. h. dass wann immer ich innerhalb einer Funktion eine Variable verändern möchte, dies zwar (bedingt, siehe weiter unten) möglich ist, diese aber nicht global verändert wird. Verwenden wir für ein Beispiel obige Funktionen:

```
#include <stdio.h>


void constint (const int a)
{
  /* a = a * 50; nicht möglich */
  printf („%d", a);
}

void normalint (int a)
{
  a = a * 100;
  printf („%d", a);
}

main()
{
  constint(10);
  normalint(10);
}
```

Bildschirmausgabe:

```
10
1000
```

Im Beispiel verwenden wir die Funktionen **constint** und **normalint**. Wie schon im Beispiel zu sehen ist, kann der Wert der Variablen a nicht verändert werden, da sie in der sog. **Parameterliste** als konstant deklariert wurde (Der Compiler bringt hier einen Fehler, deshalb ist diese Zeile auskommentiert worden, sonst würde sich das Programm überhaupt nicht starten lassen). Anders aber bei der Funktion **normalint**. Hier kann der Wert innerhalb der Funktion verändert werden. Was passiert aber bei folgendem Beispiel:

```
#include <stdio.h>


void constint (const int a)
{
  /* a = a * 50; nicht möglich, da konstant */
  printf („%d", a);
}

void normalint (int a)
{
  a = a * 100;
  printf („%d", a);
}

main()
{
  int b = 10;

  constint(b);
  normalint(b);

  printf ("%d",b) ;
}
```

Bildschirmausgabe:

```
10
1000
10
```

Wie auch im vorhergehenden Beispiel kann bei **constint** der Wert der Variablen nicht verändert werden. Bei **normalint** wird jedoch innerhalb der Funktion die Variable verändert und der Wert ausgegeben. Da die Variable jedoch verändert wird, sollte man erwarten dass bei einer nochmaligen Ausgabe der Wert auch erhalten bleibt. Dies ist jedoch nicht der Fall. Der Grund darin liegt, dass wie oben schon erwähnt bei Funktionen Variablen nur als Kopie übergeben werden und nicht die eigentliche Variable. Somit operiert die Funktion nicht auf der übergebenen Variable sondern auf einer eigens dafür angelegten Variablen. Deshalb kann. Man kann dies vielleicht so verdeutlichen:

```
#include <stdio.h>


void normalint (const int a)
{
  int kopie = a;

  kopie = kopie * 100;
  printf („%d", kopie);
}

main()
{
  int b = 10;

  normalint(b);
}
```

Will man aber nun innerhalb einer Funktion etwas verändern, so gibt es zwei Möglichkeiten.

1. Man deklariert die Funktion mit einem Rückgabewert
2. Man deklariert in der Parameterliste der Funktion einen Zeiger

```
#include <stdio.h>


int quadrat (const int a)
{
  return (a * a);
}

void quadratzeiger (int *a)
{
  *a = (*a) * (*a);
}

main()
{
  int b = 10;

  printf („%d", quadrat(10));

  quadratzeiger(&b);
  printf („%d", b) ;
}
```

Bildschirmausgabe:

```
100
100
```

Im obigen Beispiel sind nun die zwei Möglichkeiten aufgeführt:

- Die Funktion **quadrat** wurde so deklariert, das sie nach der Berechnung des Quadrates einfach den errechneten Wert zurückgibt. Dies dürfte für das Verständnis nicht weiter problematisch sein.
- Anders ist dies bei der Funktion **quadratzeiger**. Hier wird ein Zeiger in der Parameterliste deklariert. Dies haben wir im obigen Abschnitt, wo wir die Zeiger behandelten kennengelernt. Wird die Funktion dann auf-

gerufen muss man dann nur die Adresse der Variablen angeben (hier: &b). Somit kann man auch auf externe Variablen zugreifen, da hier direkt die Speicherstelle übergeben wird und keine Kopie der Speicherstelle. Somit kann man relativ einfach den Wert der Variablen verändern. Man muss innerhalb der Funktion nur berücksichtigen, dass man bei der Verwendung des Zeigers den Dereferenzierungsoperator * nicht vergisst.

Dies ist die Analogie zu der Verwendung von printf und scanf. Bei printf muss kein Wert innerhalb der Funktion geändert werden, also kann man den Wert als Kopie übergeben (call by value, es wird nur der Wert übergeben). Bei scanf muss aber die Variable verändert werden und somit muss ein Zeiger auf die Variable übergeben werden (call by reference, es wird eine Referenz, ein Verweis, auf die Variable übergeben, die somit verändert werden kann). Bleibt nur noch die Frage warum, man die Veränderung von Werten mit Zeigern und nicht Rückgabewerten von Funktionen macht, denn das wäre doch viel einfach. Natürlich, aber es gibt auch Fälle bei denen mehr Variablen verändert werden sollen und hierbei genügt nur ein Rückgabewert nicht.