# Pointer – überall Pointer

von Jörg Jacob, Mai 2004
(Version 0.2)

# Inhaltsverzeichnis



VERSIONEN:

Version 0.1 vom 28.05.04 – Erstellung
Version 0.2 vom 30.05.04 – Kosmetik

# 0. Die Zutaten

Es geht in diesem Dokument um Pointer und zwar um Pointer in der Programmiersprache C. Pointer werden im Deutschen auch oft als Zeiger bezeichnet. Zeiger ist ein Synonym für Pointer. Aber nun erstmal zu unseren Zutaten:

ein paar Variablen-Deklarationen (`int`, `short`, `char`, `float, ...`)
viele, viele Sterne `*`
einige kaufmännische Unds `&`

Das wars schon. Mehr braucht man nicht. Sieht noch alles ganz einfach aus, isses dann aber doch nicht.

# 1. Allgemeines

Nun, was machen Pointer. Pointer tun eigentlich nichts anderes, als auf einen anderen Bereich im Speicher zu zeigen. Mehr nicht. Hört sich komisch an, is aber so. Wozu das gut ist, hören wir später. Zudem weiß der Pointer aufgrund seines Typs, was der Inhalt des Speichers ist, auf den er zeigt. Ein `char` Pointer zeigt zum Beispiel (fast) immer auf einen `char` (z. B. auf den ersten Buchstaben eines Strings). Ein `int` Pointer zeigt (fast) immer auf einen `int`. Und der Pointer muss auch wissen, worauf er zeigt, weil das später für die Pointerarithmetik entscheinend wichtig ist.
Aber jetzt erst einmal was relativ einfaches. Was bewirken die `******` und die `&&&&&&&` und wie werden sie eingesetzt. Nun, da kommt schon das erste Problem. Der `*` wird einmal dazu benutzt, um Pointer zu deklarieren, aber ein anderes mal, um den Pointer zu dereferenzieren (quasi auf den Inhalt des Speichers zuzugreifen, auf den eben der Pointer zeigt). Beide Fälle bedeuten etwas Grundverschiedenes und das sollte man immer im Hinterkopf behalten.

# 2. Die ersten Pointer

Ok, jetzt deklarieren wir einfach mal unseren ersten Pointer:

```
1   char *p;
```

(Übrigens ist es Gleichgültig, ob man char* c oder char *c schreibt)

```
1   /* Erzeugung von drei Pointern auf char */
2   char *p1, *p2, *p3;
3
4
5   /* Hier werden nicht etwa drei Pointer auf char erzeugt, */
6   /* sondern ein Pointer auf char und zwei "normale" chars */
7   char* p1, p2, p3;
8   /* p2 und p3 sind keine Pointer */
```

Geschafft. War doch gar nicht schwer, oder? Nun gut, viel anfangen können wir noch nicht mit dem guten Pointer *p* (der Pointer ist *p* und nicht **p*! Der `*` wird erst einmal nur für die Deklaration gebraucht). Also, wir haben jetzt einen Pointer *p*. Aber wo zeigt er hin? Tja, das kann wohl nur der liebe Computer Gott mit Sicherheit sagen. Weil ein Pointer wird, wenn er erstellt wird, nicht auf einen definierten Wert gesetzt. Sprich, das was an der Speicherstelle zufällig (möglicherweise von einem vorherigen Programm) gerade da stand, steht jetzt in unserem Pointer drin und wird als Speicheradresse interpretiert. Im

glücklichsten Fall handelt es sich um eine Speicheradresse in unserem geschützten Speicherraum. Im schlechteren Fall zeigt er irgendwo in den Speicher, wo wir mit unserem Programm nicht zugreifen dürfen. Würden wir also den Pointer jetzt verwenden (also auf den Inhalt der Speicheradresse zugreifen, auf den er zeigt), könnte es zu eine Speicherverletzung mit Programmabbruch kommen. Also grundsätzlich sollte man Pointer initialisieren. Sprich dem Pointer die Adresse geben, worauf er zeigen soll. Im Moment wissen wir noch nicht, wo er hinzeigen soll, das kommt erst später, also lassen wir ihn der Einfachheit halber auf `NULL` zeigen.

```
1   char *p = NULL;
```

Ok, wenn wir jetzt auf die Adresse zugreifen würden, auf die der Pointer zeigt, würde es immer noch zu einer Zugriffsverletzung kommen, denn die ersten paar Bytes im Speicher (also schon etwas mehr) gehören dem Kernel und da haben wir nichts rumzupfuschen. Aber jetzt können wir eine elegante Sicherung einbauen, um so etwas zu verhindern: ein einfaches `if`:

```
1   if( p )
2   {
3    ...
4   }
```

(gleichbedeutend mit `if( p != NULL )`)

Der `if` Block wird nur ausgeführt, wenn der Pointer auf einen Speicherbereich ungleich 0 (`NULL`) zeigt.

```
1   if( !p )
2   {
3    ...
4   }
```

(gleichbedeutend mit `if( p == NULL )`)

Der `if` Block wird nur ausgeführt, wenn der Pointer auf `NULL` zeigt.

## 3. Zugriff auf Pointer

So. Wir haben immer noch nichts, was wir mit dem Pointer so recht machen können. Nun ja, bevor wir richtig loslegen noch eine Kleinigkeit, die mehr oder weniger nützlich ist. Die Ausgabe des Pointers mittels `printf()`. Aber zuerst mal vorweg: Jeder Pointer, egal auf welchen Variablentyp er auch zeigen mag, hat im Speicher immer die selbe Größe, bei unseren Intelmaschinen im Normalfall 4 Byte. Also ein `char*` ist genauso groß wie ein `double*` oder ein `void*` (ja, das gibt es auch, aber mehr dazu später). Auch vom Inhalt sind alle diese Pointer gleich: sie beinhalten eine Speicheradresse. Also braucht man nur eine Art, um Pointer auszugeben, und man muss nicht, wie bei normalen Variablen unterscheiden, ob es nun ein `int` ist (`%i`) oder ein `float` (`%f`). `printf`-Symbol `%p`:

```
1   printf( "%p\n", p );
```

Und hier mal ein ganzes Beispielprogramm:

```
1   #include <stdio.h>
2
3   int main( void )
4   {
5       char* p = NULL;
6       char* p2;
7
8       printf( "%p\n", p );
9       printf( "%p\n", p2 );
10
11      return 0;
12  }
```

Die Ausgabe von *p* variiert von Compiler zu Compiler bzw. von Betriebssystem zu Betriebssystem. Mal wird *(nil)* ausgeben, mal 00000000. Was bei *p2* rauskommt, ist davon abhängig, was vorher an dem Speicherbereich gestanden haben mag - möglicherweise 0x123F093A oder auch CCCCCCC. Die Darstellung ist wieder Compilerabhängig. Das Ergebnis ist also bei *p2* nicht definiert (s.o.). Aber warum können wir denn jetzt doch auf den Pointer zugreifen und ihn ausgeben? Oben stand doch was anderes? Die Antwort ist recht einfach: Wir greifen ja jetzt nur auf den **Inhalt** des Pointers zu, sprich auf die Zahl, die er in **seinem** Speicherbereich gespeichert hat und nicht auf die Speicheradresse, die dieser Zahl entspricht. Würden wir jetzt auf die Speicheradresse zugreifen, auf die *p* zeigt, würde es zu einem Speicherzugriffsfehler kommen. Wie genau man zugreift kommt später.

# 4. Verwendung von Pointern

So, jetzt haben wir schon etliche Pointer angelegt, aber noch immer nichts wirklich sinnvolles gemacht. Das kommt jetzt:

```
1   char c = 'A';
2   char *p = NULL;
3   p = &c;
4   printf( "Der Inhalt von c: %c\n", c );
5   printf( "Der Inhalt von p: %p\n", p );
6   printf( "Die Speicheradresse von c: %p\n", &c );
7   printf( "Inhalt der Adr. von c über Pointer p: %c\n", *p );
```

(Statt der zwei Zeilen **`char *p = NULL;`** und **`p = &c;`** kann man auch schreiben **`char *p = &c`**)

So, jetzt kommen wir zu den vielen neuen Sachen. Also in der ersten Zeile wird eine normale `char` Variable angelegt und ihr wird als Inhalt der Großbuchstabe *A* zugewiesen. Das sollte noch einigermaßen klar sein. Auch die nächste Zeile (2) ist nichts wirklich Neues: Es wird ein Pointer mit dem Namen *p* angelegt und ihm wird als Inhalt `NULL` zugewiesen, sprich er *zeigt* gerade auf `NULL`.
Jetzt wird es Interessant. Die nächste Zeile (3) ist was Neues. Die sieht auch ganz schön komisch aus, ist aber richtig so. Was passiert? Also ganz einfach ausgedrückt wird der Pointervariable *p* etwas zugewiesen. Das kennen wir schon. Also der Pointervariablen *p* wird ***&c*** zugewiesen. Ja, schön, oder? Nein, eigentlich nicht schön. Was soll dieses komische kaufmännische UND vor dem *c*? Zur Erklärung: Das **&**-Zeichen liefert, vor einem Variablennamen platziert, die Adresse eben dieser Variablen im

Speicher zurück. Und da sind wir doch schon da, wo wir hin wollten: der Pointer *p* erwartet ja gerade eine Adresse im Speicher und die bekommt er jetzt auch und zwar eben die Adresse im Speicher, an der *c* abgelegt ist. Schön ;-). Also noch mal: Mit der Zeile `p = &c;` wird dem Pointer *p* die Speicheradresse, an der der `char` *c* abgelegt ist, zugewiesen. Klar?

Ok, weiter im Programmtext: Die erste `printf()`-Zeile (4) sollte kein Problem sein; hier wird einfach die Variable *c* (sprich deren Inhalt) ausgegeben. Die nächste Zeile (5) ist schon etwas interessanter, aber kennen tun wir sie auch schon. Hier wird der Inhalt unseres Pointers ausgegeben. Dieser ist jetzt im Moment die Speicheradresse an der *c* im Speicher liegt. Die Zeile 6 gibt nun *&c* aus. *&c* gibt ja die Adresse im Hauptspeicher zurück, an der *c* liegt. Also gibt dieses `printf()` aus Zeile 6 genau dieselbe Zahl wie das `printf()` aus Zeile 5 aus. Nun kommt was wirklich Neues in Zeile 7: ein `printf ( "....%c\n", *p );`! Wie jetzt? Wieso setzt der Typ jetzt hier ein `*` vors *p*? Also, wenn wir auf *p* zugreifen würden und keinen Stern davor setzen würden, würden wir ja den Inhalt von *p* erhalten, den wir aber jetzt gerade nicht wollen. Wir wollen nicht den Inhalt von *p* haben, sondern den Inhalt des Speicherbereichs, auf den *p* gerade zeigt (ist in unserem Fall der Speicherbereich - 1 Byte - an dem *c* abgelegt ist). Und das schaffen wir mit dem `*`-Operator oder auch Dereferenzierungsoperator genannt. Also, mit dem Stern außerhalb von Variablendeklarationen dereferenziert man einen Pointer. Hört sich komisch an, ist aber so. Das hier der Operator `*` verwendet wurde, macht das Ganze sehr unübersichtlich, da der Stern ja schon andere Bedeutungen hat (als Multiplikationszeichen, Pointerdeklartionszeichen etc.), aber da gibt es keine andere Möglichkeit. Noch einmal eine Zusammenfassung:

| | |
|---|---|
| `char c ='A';` | Deklaration einer Charakter Variable und Zuweisung von 'A' |
| `char *p;` | Deklaration eines Pointers |
| `p = &c;` | Zuweisung einer Speicheradresse an den Pointer |
| `c = *p;` | Dereferenzierung des Pointers und Zuweisung des entsprechenden Speicherinhalts an c |

## 5. Sinn von Pointern

So, toll, oder? Aber was haben wir jetzt davon? Wir haben eine Variable *c* und einen Pointer *p* mit denen wir eigentlich genau dasselbe machen können, nämlich auf den Inhalt von *c* zugreifen. Wo liegt der tiefere Sinn? Das ist jetzt wahrscheinlich das Schwierigste an der ganzen *Rumpointerei*, das zu erklären. Es gibt viele Dinge, die man ohne Pointer einfach nicht erledigen kann. Zum Beispiel Arrays an Funktionen zu übergeben, dynamischen Speicher allokieren oder auch um überhaupt mit Arrays zu arbeiten. Was erzählt denn der da wieder für einen Mist? Ich brauche doch keine Pointer, um mit Arrays zu arbeiten! Ich benutze doch immer die schönen Indexe `[x]`. Nun ja, ich möchte euch nicht erschrecken, aber die Index-Klammern sind eigentlich nur eine vereinfachte Schreibweise für Pointerarithmetik. Uff - setzen lassen - Beispiel:

```
1   char meinstring[] = "Hallo!";
2   printf( "Erster Buchstabe: %c\n", meinstring[0] );
```

Andere Schreibweise:

```
1   char* meinstring = "Hallo!";
2   printf( "Erster Buchstabe: %c\n, *(meinstring+0) );
```

(das +0 könnte man weglassen, aber hier bleibt es wegen dem Verständnis da).

**SCHOCK**. Was macht der da. Was schreibt der da hin? Tjo, eigentlich schreibe ich genau das Gleiche hin wie vorher, nur ein bisschen anders. Also, der Name eines Arrays (hier *meinstring*) ist nichts anderes als ein Pointer, der als Inhalt die Adresse des ersten Elements des Arrays (bzw. bei `char`-Array-Strings den ersten Buchstaben des Strings) enthält. Hart aber wahr. Also, um auf die einzelnen Elemente eines Arrays (oder im speziellen: eines Strings) zuzugreifen benutzt man überlicherweise die beiden eckigen Klammern `[` und `]`. In diesen gibt man den Index an, den man sich anschauen möchte. Intern arbeitet das Ganze mit Pointerarithmetik.

Jetzt am konkreten Beispiel unseres Strings: Wir nehmen uns den Namen des Strings *meinstring*, welcher ja, wie wir gehört haben, ein Pointer auf das erste Element desselbigen ist und zählen die Zahl dazu, dessen Index wir haben möchten: *meinstring+n* (in unserem Beispiel +0). Nun haben wir automatisch die Speicheradresse des n-ten Elements des Arrays. Aber wir wollen ja nicht die Speicheradresse haben, sondern den Inhalt. Wie kommt man von Speicheradressen auf den Inhalt? Ganz einfach und kennen wir auch schon: durch Dereferenzierung. Also wir brauchen den `*`. Allerdings ist der Stern von der Priorität höher als das +. Das + muss allerdings vorher ausgeführt werden, weswegen wir ein bisschen Klammern müssen: *`*(meinsteing+n)`*. Wir bewegen uns also um n Stellen in unserem Array vorwärts, derefenzieren und haben damit den Inhalt der Speicherstelle. Schön. Im übrigen überwacht kein Mechanismus, ob ihr auf diese Weise schon über das letzte Element eures Arrays hinausgegangen seid oder nicht. Sprich hier ist wieder die Gefahr, einen Speicherzugriffsfehler zu verursachen recht groß. Bei ordentlich terminierten Strings sollte man spätestens bei `\0` aufhören zu lesen. Der Rest dahinter mag noch zum Array-Speicherbereich gehören, ist allerdings eh uninteressant. Bei anderen Arrays (z. B. `int`-Arrays) muss man sich selbst darum kümmern, nicht über die Grenzen hinauszugehen.

Bleiben wir noch mal bei unseren Arrays. Wir haben vorhin mit unseren `char`-Arrays gearbeitet und wie wahrscheinlich viele schon wissen, ist ein `char` immer 1 Byte groß. Also ist es kein Problem mit der Pointerarithmetik. Wenn ich das 5. Zeichen haben möchte (welches ja an Position 4 im Array steht, weil die Zählung ja bei 0 beginnt), zähle ich einfach 4 zu meinem Pointer dazu. Aber wie ist es z. B. mit `int` Arrays? Wir wissen ja, das ein `int` auf einer 32 Bit Maschine 4 Byte lang ist. Muss man jetzt 4 dazuzählen, um aufs nächste Element zu kommen? Nein.

```
1   int array[] = { 11, 12, 13, 14 };
2
3   // wir wollen die 14
4   printf( "%i\n", array[3] );
5   // bzw
6   printf( "%i\n", *(array+3) );
```

Glück gehabt! Wir brauchen uns also bei Pointerarithmetik keine Gedanken zu machen, auf welchen Inhalt unser Pointer denn zeigt. Aber warum? Woher weiß der Pointer, dass er bei `char` jeweils um 1 Byte weiter im Speicher gehen muss, aber bei `int` um 4 Byte? Ganz einfach: Wenn wir einen Pointer anlegen, geben wir ihm den Typ des Speicherinhalts, auf den er zeigt gleich mit. Wir legen ja einen

`char* p;` bzw. `int* p;` an. Sprich aufgrund dieser Vereinbarung weiß der Compiler, um wie viel Byte er den Pointer im Speicher vorwärtsbewegen muss. Allerdings hat das auch so seine Tücken: Man kann auch einen `int` Pointer auf ein `char`-Array Zeigen lassen. Allerdings möchte ich darauf erst mal nicht näher eingehen.

## 6. Funktionen mit mehreren Rückgabewerten

So, Pointer und Arrays abgehakt. Jetzt geht es weiter. Ein weiterer Nutzen von Pointern ist, wenn man von Funktionen mehr als ein Ergebnis zurückhaben möchte. Im Normalfall kann eine Funktion ja nur ein Ergebnis zurückliefern. Was machen wir nun, wenn wir mehr als ein Ergebnis von der Funktion benötigen. Probieren wir es einfach mal. Wir wollen von zwei Zahlen die Summe bzw. die Differenz von einer Funktion berechnen und zurückgeben lassen:

```
1   int rechne( int a, int b );
2
3   int main( void )
4   {
5       int zahl1 = 4;
6       int zahl2 = 8;
7       int ergebnis1;
8       int ergebnis2;
9
10      ergebnis1 = rechne( zahl1, zahl2 );
11
12      return 0;
13  }
14
15  int rechne( int a, int b )
16  {
17      int e1 = a + b;
18      int e2 = a - b;
19
20      // wir können nur ein Ergebnis zurückgeben
21      return e1;
22  }
```

Tjo, also hier sind wir an der Grenze der "normalen Funktionen" angelangt. Wir wollen zwei Ergebnisse haben, aber die Funktion kann nur einen Wert zurückgeben. Wie lösen wir das Problem? Ganz einfach: Mit Pointern:

```
1   void rechne( int z1, int z2, int* erg1, int* erg2 );
2
3   int main( void )
4   {
5       int zahl1 = 4;
6       int zahl2 = 8;
7       int ergebnis1;
8       int ergebnis2;
9
10      rechne( zahl1, zahl2, &ergebnis1, &ergebnis2 );
11
12      return 0;
13  }
14
15  void rechne( int z1, int z2, int* erg1, int* erg2 )
16  {
17      *erg1 = z1 + z2;
18      *erg2 = z1 - z2;
19  }
```

Nun haben wir die Lösung. Wir übergeben der Funktion *rechne()* nicht nur die zwei Zahlen, mit denen wir rechnen wollen, sondern auch zwei Pointer auf zwei `int` Variablen, in denen wir das Ergebnis dann speichern möchten.
In *rechne()* dereferenzieren wir *erg1* und *erg2* (sprich, wir greifen direkt auf den Speicherbereich zu) und speichern das Ergebnis von *z1 + z2* bzw. *z1 - z2* in diesem Speicherbereich. Nachdem die Funktion *rechne()* beendet ist, sind die Variablen *z1* und *z2* sowie *erg1* und *erg2* wieder aus dem Speicher entfernt (zumindest theoretisch). Die Adressen von *ergebnis1* und *ergebnis2* wurden allerdings an die Funktion übergeben und somit hat die Funktion die Speicherbereiche verändert, an denen *ergebnis1* und *ergebnis2* im Speicher abgelegt sind. Somit haben wir nach *rechne()* die Ergebnisse da, wo wir sie haben wollen. Einfach mal ein bisschen rumprobieren.

# 7. Pointerarithmetik

So, wenden wir uns kurz der Pointerarthimetik zu. Was dürfen wir alles mit Pointern rechnen? Also, wie wir schon gesehen haben, dürfen wir Ganzzahlen (Integer) zu Pointern dazu**addieren**. Genauso dürfen wir Ganzzahlen von Pointern **subtrahieren** und auch den **Inkrementopertor** `++` und den **Dekrementoperator** `--` auf Pointer anwenden. Was man noch darf, ist zwei Pointer voneinander zu **subtrahieren**; das Ergebnis ist der Abstand der beiden Pointer im Speicher (bei Pointern, die auf das gleiche Array zeigen: die Anzahl der Elemente zwischen den Pointern). Es ist zu beachten, dass die **Addition** und **Subtraktion** von Ganzzahlen sowie die **Differenz** zweier Pointer die Pointer selbst nicht verändern. Der **Inkrement**- bzw. **Dekrementoperator** allerdings schon. Sprich der Pointer zeigt nach `++` bzw. `--` auf eine andere Speicherstelle als zuvor. Aber hier einmal ein einfaches Beispiel:

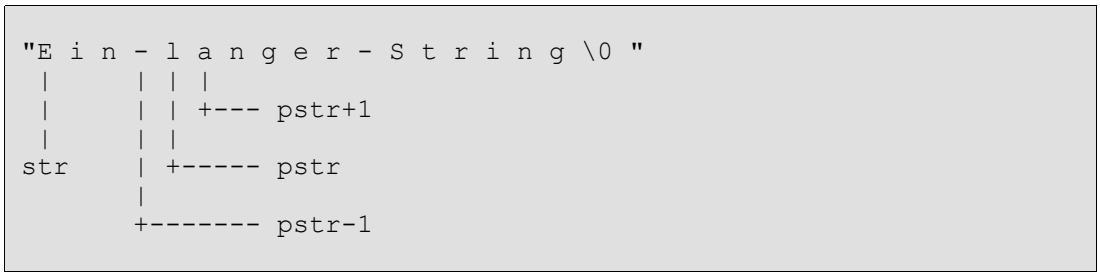
```
1   #include <stdio.h>
2
3   int main( void ) {
4       char* str = "Ein-langer-String";
5       char* pstr = str+4;
6
7       printf( "str: %s\npstr: %s\n", str, pstr );
8
9       printf( "+1 zu pstr Buchstabe: %c\n", *(pstr+1 ) );
10      printf( "-1 ab pstr Buchstabe: %c\n", *(pstr-1 ) );
11
12      pstr++;
13      printf( "pstr nach ++: %c\n", *(pstr) );
14
15      pstr--;
16      printf( "pstr nach --: %c\n", *(pstr) );
17
18      printf( "Abstand pstr und str: %i\n", pstr - str );
19
20      return 0;
21  }
```

Ausgabe:

```
str: Ein-langer-String
pstr: langer-String
+1 zu pstr Buchstabe: a
-1 ab pstr Buchstabe: -
pstr nach ++: a
pstr nach --: l
Abstand pstr und str: 4
```

Nun zur Erklärung: Zeile 4 ist klar; wir legen ein `char`-Array mit dem Inhalt "*Ein-langer-String*" an. In Zeile 5 wird ein `char` Pointer *pstr* angelegt, der die Adresse *str+4* übergeben bekommt, also 4 Byte im Speicher weiter als *str* zeigt.
In der Zeile 7 werden die beiden Pointer als Strings an `printf()` übergeben. Und siehe da, *str* hat als Inhalt "*Ein-langer-String*", während *pstr* "*langer-String*" hat, eben 4 Byte weiter als *str*. Im übrigen ist der Text "*Ein-langer-String*" nur einmal im Speicher vorhanden; wir haben ausschließlich zwei Pointer, die

auf verschiedene Speicherbereiche eben dieses Strings zeigen. Sollte über *pstr* der String verändert werden, würde natürlich bei der Ausgabe von *str* auch ein veränderter String rauskommen. Nun kommt die eigentlich richtige Arithmetik:
In Zeile 9 wird *pstr+1* als `char` ausgegeben. Es wird ein *a* ausgegeben. *pstr* zeigt ja auf den ersten Buchstaben eines `char` Arrays. In unserem Fall ist der erste Buchstabe das *I*. *pstr+1* gibt *a* heraus, was auch stimmt, weil auf das *I* ein *a* folgt.
In Zeile 10 wird *pstr-1* als `char` ausgegeben. In der Zeile oben wurde *pstr* selbst nicht verändert, also *pstr* zeigt weiterhin auf das *I*. *pstr-1* wäre also der Buchstabe vor dem *I*, eben das -.
Noch immer haben wir *pstr* selbst nicht verändert. Nun Inkrementieren wir *pstr* in Zeile 12 und ab nun zeigt der Pointer nicht mehr auf die Speicherstelle, in der der Buchstabe *I* gespeichert ist, sondern eine Speicherstelle weiter auf das *a*. Das `printf()` aus Zeile 13 gibt also *a* aus.
Einmal dekrementieren aus Zeile 15 bringt *pstr* wieder zurück auf die Speicherstelle, in der das *I* gespeichert ist.
Das letzte was wir durchführen, ist eine Subtraktion zweier Pointer. Das Ergebnis ist eine Ganzzahl (Integer), der den Abstand der beiden Pointer zurückgibt. Da *pstr* 4 Einheiten von *str* entfernt ist, wird hier 4 ausgegeben.



# 8. Pointerpointer

So, hier sei einmal der Hinweis gegeben, dass es natürlich auch Pointerpointer gibt, sprich Pointer, die ihrerseits wieder auf einen Pointer zeigen. Hier ein kleines Beispiel:

```
1   // Ein char, der als Inhalt ein A hat
2   char c = 'A';
3
4   // Ein Pointer auf char, der als Inhalt die Adresse von c hat
5   char *p = &c;
6
7   // Ein Pointerpointer auf char, der als Inhalt die Adresse von p hat
8   char **pp = &p;
9
10  // drei Wege, um das A auszugeben:
11  printf( "%c\n", c );
12  printf( "%c\n", *p );
13  printf( "%c\n", **p );
```

Ok, Pointerpointer braucht man z. B., wenn man einen Pointer an eine Funktion übergeben will und man innerhalb der Funktion den Inhalt des Pointers (sprich die Speicheradresse, auf die er zeigt) verändern möchte. Übergäbe man in diesem Fall nur einen einfachen Pointer an die Funktion, würde man zwar den Inhalt des Speicherbereichs verändern können, auf den der Pointer zeigt, allerdings nicht den Pointer selbst, da dieser "by value" übergeben wird.
Ein Beispiel. Zwei Pointer zeigen auf zwei Strings. Wir wollen die Pointer vertauschen:

```
1   #include <stdio.h>
2
3   void swp( char** str1, char** str2 );
4
5   int main( void ) {
6       char* str_eins = "Hallo123";
7       char* str_zwei = "Was anderes";
8
9       printf( "str_eins: %s\n", str_eins );
10      printf( "str_zwei: %s\n", str_zwei );
11
12      swp( &str_eins, &str_zwei );
13
14      printf( "str_eins: %s\n", str_eins );
15      printf( "str_zwei: %s\n", str_zwei );
16
17      return 0;
18  }
19
20  void swp( char** str1, char** str2 )
21  {
22      char* tmp;
23
24      tmp = *str1;
25      *str1 = *str2;
26      *str2 = tmp;
27  }
```

Ok, warum übergeben wir nun Pointerpointer anstatt normale Pointer an die Funktion *swp()*. Der Grund ist, wie oben schon beschrieben, dass wir den Inhalt (sprich die gespeicherte Adresse), die in *str_eins* bzw. *str_zwei* enthalten ist, verändern wollen. Würden wir an *swp()* nur normale `char`-Pointer übergeben, würden wir die Adresse aus *str_eins* bzw. *str_zwei* "by value" bekommen, also als Wert. Die Adressen von *str_eins* und *str_zwei* allerdings nicht; so könnten wir diese auch nicht ändern. Lösung: Wir lassen uns die Adressen von *str_eins* und *str_zwei* übergeben, also die Zahlen, wo eben diese im Speicher abgelegt sind. Wir legen eine Hilfsvariable *tmp* an und speichern in dieser den Wert, der in *str_eins* gespeichert ist. An diesen kommen wir durch einmal dereferenzieren heran, weil ja der Inhalt des Pointerpointers die Speicheradresse von *str_eins* ist. Dann weisen wir **str1* noch **str2* zu. Und am Schluss noch die kurz gemerkte *tmp*-Variable wieder **str2*.
Siehe da, wir haben die Strings vertauscht. Im Speicher stehen sie natürlich immer noch an der selben Stelle, die Pointer zeigen nur jeweils auf den anderen. Und hier gleich noch ein riesiger Vorteil von Pointern: **Geschwindigkeit**! Es ist um Welten schneller, nur die Pointer zu vertauschen als den kompletten String: Im Falle von zwei Strings *str1* und *str2* müssten wir bei einem kompletten Tausch erst einmal einen Hilfsstring *tmp* anlegen, der den Inhalt von *str1* fassen kann. Dann müssten wir `char` für `char` str1 nach *tmp* kopieren, dann `char` für `char` von *str2* zu *str1* und dann noch mal `char` für `char` von *tmp* nach *str2*. Es ist offensichtlich, dass diese Speicherumschaufelei deutlich mehr Zeit in Anspruch nimmt, als einfach die Pointer zu vertauschen. Geschweige denn die Problematik mit der Länge der Strings und dem zu Verfügung stehenden Speicher.

# 9. Funktionspointer

Mmmh, da fällt mir noch was ein. Funktionen sind ja soweit bekannt. So kann z. B. eine Funktion aussehen:

```
1   void foo( void )
2   {
3      //.... Funktionsrumpf
4   }
```

Ok, soweit klar. Die Funktion ist eigentlich eine Prozedur, weil sie keinen Wert zurückliefert. Sie ist ja `void`, aber das ist für uns jetzt völlig unerheblich. Also weiter im Text: Die Funktion bekommt auch keine Parameter übergeben. Das ist auch nicht weiter interessant und der Funktionsrumpf interessiert uns jetzt auch nicht weiter. Was bleibt übrig? Genau, der Name der Funktion: *foo*. "Oh toll!", werdet ihr denken, "der Name, über den wissen wir doch schon alles. Den schreiben wir hin, wenn wir die Funktion aufrufen wollen." - Richtig, da habt ihr völlig recht, aber der Name ist eigentlich noch etwas anderes. Der Name einer Funktion kann in unserem Programm wie ein Pointer benutzt werden, toll, oder? ;-) Der Name einer Funktion ist ein Pointer, der die Anfangsadresse unsere Funktion enthält. Würden wir also folgendes schreiben, bekämen wir die Speicheradresse unsere Funktion zurückgegeben:

```
1   printf( "%p\n", foo );
```

Ok, was bringt uns das jetzt? Ja, eine ganze Menge. Mit dem Wissen, dass Funktionsnamen eigentlich Pointer auf den Anfang der Funktion sind, haben wir die Möglichkeit, diese Pointer anderen Funktionen zu übergeben, damit diese sie ausführen können. Schön - ich rede gar nicht lang rum, sondern schreib einfach mal einen Beispielcode hin:

```
1   #include <stdio.h>
2
3   void foo( void (*func)(void) );
4
5   void func1( void );
6   void func2( void );
7   int  func3( int i );
8
9   int main( void ) {
10      foo( func1 );
11      foo( func2 );
12      foo( ( void* )func3 );
13
14      return 0;
15  }
16
17  void foo( void (*func)(void) )
18  {
19      (*func)();
20  }
21
22  void func1( void )
23  {
24      printf( "Ich bin Funktion I\n" );
25  }
26
27  void func2( void )
28  {
29      printf( "Ich bin Funktion II\n" );
30  }
31
32  int func3( int i )
33  {
34      printf( "Ich bin Funktion III\n" );
35      return 0;
36  }
```

Gründgüter! Was ist das??? Ja, das ist jetzt wirklich was feines. Das ist wieder eine Sache, die C so unheimlich mächtig und zugleich so unheimlich Fehleranfällig macht. Hier werden nun keine normalen Werte mehr an die Funktion übergeben, sondern Funkionspointer. Fein. Aber hier die konkrete Erklärung des Codes:

Zeile 1: Ein `include` für `printf()`

Zeile 3: Hier ist unsere Prototyp für die Funktion, die einen Funktionspointer übergeben bekommen soll. Interessant hierbei ist für uns ausschließlich dieser komische Übergabeparameter *`void (*func)(void)`*. Was sagt uns dieser Parameter. Erst mal, es handelt sich nur um einen einzigen Parameter, nicht etwa um mehrere. Dieser Parameter sagt eigentlich nicht mehr, als das die Funktion *foo()* als Übergabe einen Pointer auf eine Funktion erwartet (**func*), die den Rückgabewert `void` hat und selbst als Übergabeparameter `void` erwartet. Ok, bei dem Rückgabewert können wir später mittels Typecasting schön tricksen, aber mehr dazu später. Wir können auch Pointer auf Funktionen übergeben, andere Parameter als `void` haben, allerdings können wir diese Parameter beim Aufruf mittels des Pointers nicht mehr an die Funktion übermitteln.

Zeile 5-7: Ok, hier sind die Prototypen der Funktionen für unseren Test. Zwei, so wie sie die Funktion *foo* haben möchte, eine mit einem anderen Rückgabewert und anderen Parameter.

Zeile 10-12: Hier innerhalb der `main`-Funktion wird ausschließlich die Funktion *foo()* aufgerufen. Als Parameter erhält sie die Pointer auf unsere Funktion. Beim dritten Aufruf müssen wir in `void*` casten, weil die Funktion *func3()* ja eigentlich eine `int` ist, und somit der Pointer auf diese Funktion ein `int*`. Deswegen das (`char*`) vor dem Parameter. Sollten wir das weglassen, würde es allerdings nur zu einem Compilerwarning kommen.

Zeile 17-20: Unsere *foo()*-Funktion. Was die Syntax des Übergabeparameters betrifft, das haben wir oben schon geklärt. Also kümmern wir uns nur noch um den Funktionsaufruf selbst. Wir haben den Pointer *func*. Um auf die entsprechende Funktion zuzugreifen, auf den *func* zeigt, müssen wir dereferenzieren, sprich einen Stern vor *func* schreiben. Zudem hat eine Funktion immer eine Parameterliste (auch wenn diese, wie in diesem Fall leer sein mag). Also kommt nach dem derefernzierten Funktionsaufruf die Parameterliste.

Rest: Dies sind ausschließlich die ausprogrammierten Funktionen, die wir aufrufen.

So, gar nicht so einfach, aber wenn ihr es euch gut anschaut und ein bisschen rumprobiert, ist selbst das nicht sonderlich schwer. Aber hier noch mal ein kleines Beispiel, wie man mit Funktionen umgeht, die einen oder mehr Übergabeparameter haben:

```
1   #include <stdio.h>
2
3   void foo( void (*func)( int, int ) );
4   int func( int a, int b );
5
6   int main( void ) {
7       foo( (void *)func );
8
9       return 0;
10  }
11
12  void foo( void (*func)( int, int ) )
13  {
14      (*func)( 2, 3 );
15  }
16
17  int func( int a, int b )
18  {
19      printf( "Ich die Funktion und a + b = %i\n", a + b );
20
21      return 0;
22  }
```

Also viel hat sich nicht geändert. Ausschließlich das `(int, int)` ist bei der Parameterliste von *foo()* dazugekommen. Man kann der Funktion weiterhin Pointer auf Funktionen übergeben, die andere Parameterlisten haben (Warnungen des Compilers werden ab und an auftauchen), allerdings sollte man das doch vermeiden. Was verpflichtend ist, ist die Mitgabe der Parameter beim Aufruf der Funktion in *foo()*. Soweit zu Funktionspointern.

## 10. Pointer als Rückgabewerte

Jetzt mal wieder etwas Einfaches: Pointer als Rückgabewert einer Funktion. Man stelle sich vor, man hat eine Funktion, die als Übergabeparameter einen Pointer auf einen String (`char` Array) hat und die Adresse des ersten in dieser Funktion vorkommenden Buchstabens *a* zurückliefern soll. Ok, ziemlich einfach: wir brauchen eine Funktion (wie nennen sie mal wieder *foo()* ) und die braucht als Parameter einen `char*` und als Rückgabewert auch einen `char*`. Hier einfach mal der Prototyp:

```
1   char* foo( char* );
```

Und hier, wie wir das Ganze ausprogrammieren würden (übrigens, für den Fall, dass in dem String gar kein kleines *a* vorhanden ist, soll die Funktion `NULL` zurückliefern):

```
1   #include <stdio.h>
2   #include <string.h>
3
4   char* foo( char* );
5
6   int main( void )
7   {
8       char str[] = "Hier ist ein a drin";
9       char* p_auf_a = NULL;
10
11      p_auf_a = foo( str );
12
13      if( p_auf_a )
14      {
15          printf( "Die Adresse von dem ersten a: %p\n", p_auf_a );
16          printf( "Das a selbst: %c\n", *p_auf_a );
17          printf( "Der String ab a: %s\n", p_auf_a );
18      }
19
20      return 0;
21  }
22
23  char* foo( char* s )
24  {
25      int i;
26
27      for( i = 0; i < strlen( s ); i++ )
28          if( s[i] == 'a' )
29              return s+i;
30
31      return NULL;
32
33  }
```

Also, ich spreche mal die wichtigen Zeilen an:
In Zeile 11 wird die Funktion *foo()* aufgerufen. Sie bekommt als Parameter den Pointer auf unseren String. Das Ergebnis der Funktion wird in dem `char` Pointer *p_auf_a* gespeichert. Wie wir wissen, liefert

unsere Funktion `NULL` zurück, wenn kein *a* in dem String gefunden wird. Also müssen wir verhindern, dass unser Pointer benutzt wird, wenn er ein `NULL`-Pointer ist. Dies machen wir mit der `if`-Abfrage in Zeile 13. Das haben wir auch schon mal ganz am Anfang gesehen, deswegen spar ich mir die Erklärung. In Zeile 15-17 werden ein paar Sachen ausgegeben, wie z. B. die Adresse, die in *p_auf_a* gespeichert ist, sowie der Speicherinhalt, auf den *p_auf_a* zeigt (eben das erste a in unserem String).
Die Funktion *foo()* selbst (ab Zeile 23) tut nun nichts anderes, als den String, den sie als Pointer übergeben bekommt von 0 bis `strlen()-1` zu durchlaufen. Wenn ein *a* gefunden wird, befindet es sich an der i-ten Position im String und da wir schon oben einiges über Pointerarithmetik gehört haben, ist das `return s+i;` mit Sicherheit für niemanden ein Problem. Aber trotzdem noch mal. *s* zeigt auf den Anfang des Strings, *i* ist die "relative" Position (incl. der 0), an der das erste *a* gefunden wurde. Also ist *s+i* die Speicheradresse, an der das erste *a* im Speicher steht. Sollte bis zum Stringende kein *a* gefunden werden, liefert die Funktion wie gefordert den Rückgabewert `NULL`.