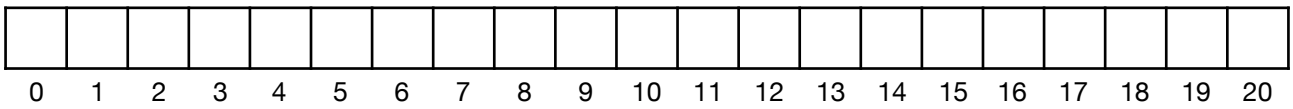
Zeiger in C

Prof. Dr. Malte Weiß

Dieser Artikel erläutert das Konzept der Zeiger in der Programmiersprache C.

Speicher als Anordnung von Zellen

Sämtliche Programme und die damit verarbeiteten Daten befinden sich zur Laufzeit eines Programms im Arbeitsspeicher. Wie wir vom Von-Neumann-Modell wissen, besteht dieser Speicher aus einer Folge einzelner Zellen:



Die Zellen sind fortlaufend durchnummeriert, wobei die Nummerierung – wie in der Informatik üblich – bei 0 beginnt. Die Nummer einer Speicherzelle bezeichnet man als **Speicheradresse**, oder einfach als **Adresse**.

Jede Speicherstelle ist genau 1 Byte – also 8 Bits – groß. Da ein Bit genau zwei mögliche Zustände (0 oder 1) besitzen kann, kann eine einzelne Speicherstelle genau 2^8, d.h. 256, unterschiedliche Werte annehmen. In jeder Speicherstelle kann demnach grundsätzlich nur einer der folgende Werten stehen.

00000000	00100000	01000000	01100000	10000000	10100000	11000000	11100000
00000001	00100001	01000001	01100001	10000001	10100001	11000001	11100001
00000010	00100010	01000010	01100010	10000010	10100010	11000010	11100010
00000011	00100011	01000011	01100011	10000011	10100011	11000011	11100011
00000100	00100100	01000100	01100100	10000100	10100100	11000100	11100100
00000101	00100101	01000101	01100101	10000101	10100101	11000101	11100101
00000110	00100110	01000110	01100110	10000110	10100110	11000110	11100110
00000111	00100111	01000111	01100111	10000111	10100111	11000111	11100111
00001000	00101000	01001000	01101000	10001000	10101000	11001000	11101000
00001001	00101001	01001001	01101001	10001001	10101001	11001001	11101001
00001010	00101010	01001010	01101010	10001010	10101010	11001010	11101010
00001011	00101011	01001011	01101011	10001011	10101011	11001011	11101011
00001100	00101100	01001100	01101100	10001100	10101100	11001100	11101100
00001101	00101101	01001101	01101101	10001101	10101101	11001101	11101101
00001110	00101110	01001110	01101110	10001110	10101110	11001110	11101110
00001111	00101111	01001111	01101111	10001111	10101111	11001111	11101111
00010000	00110000	01010000	01110000	10010000	10110000	11010000	11110000
00010001	00110001	01010001	01110001	10010001	10110001	11010001	11110001
00010010	00110010	01010010	01110010	10010010	10110010	11010010	11110010
00010011	00110011	01010011	01110011	10010011	10110011	11010011	11110011
00010100	00110100	01010100	01110100	10010100	10110100	11010100	11110100
00010101	00110101	01010101	01110101	10010101	10110101	11010101	11110101
00010110	00110110	01010110	01110110	10010110	10110110	11010110	11110110
00010111	00110111	01010111	01110111	10010111	10110111	11010111	11110111
00011000	00111000	01011000	01111000	10011000	10111000	11011000	11111000
00011001	00111001	01011001	01111001	10011001	10111001	11011001	11111001
00011010	00111010	01011010	01111010	10011010	10111010	11011010	11111010
00011011	00111011	01011011	01111011	10011011	10111011	11011011	11111011
00011100	00111100	01011100	01111100	10011100	10111100	11011100	11111100
00011101	00111101	01011101	01111101	10011101	10111101	11011101	11111101
00011110	00111110	01011110	01111110	10011110	10111110	11011110	11111110
00011111	00111111	01011111	01111111	10011111	10111111	11011111	11111111

Wenn wir in C eine Variable deklarieren, belegt sie eine oder mehrere Speicherzellen in Folge.

Datentypen

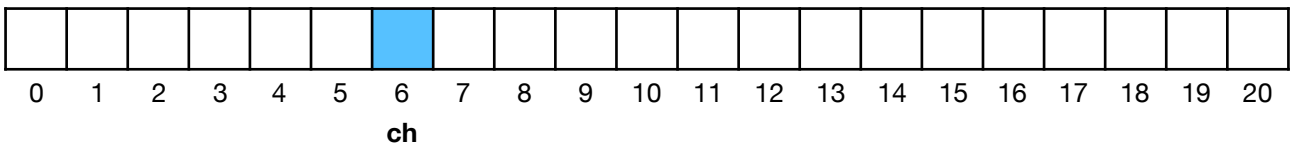
Der Datentyp einer Variablen bestimmt zwei Dinge:

1. **Wieviel Speicherplatz** eine Variable belegt.
2. Wie der belegte Speicher **interpretiert** wird.

Nehmen wir als Beispiel den Datentyp `char`. Ein `char` belegt klassischer Weise genau ein Byte. Deklarieren wir in einem C-Programm also eine Variable

```
char ch;
```

belegt sie im Speicher genau eine Speicherzelle. Das sieht im Speicher in etwa so aus:



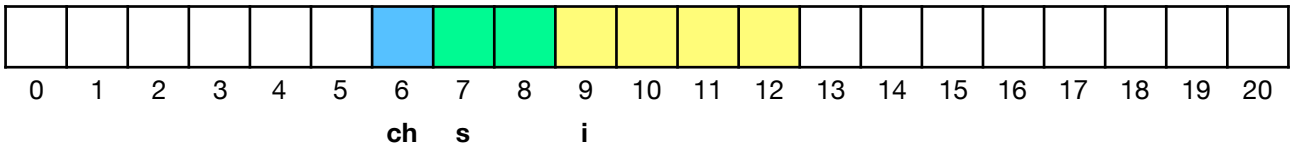
Die Variable **ch** ist in dem Beispiel in der Zelle mit der Speicheradresse 6 gespeichert.

Jetzt wollen wir in einer Variablen nicht unbedingt nur 256 Werte speichern können, sondern benötigen für die meisten Berechnungen eher einen größeren Wertebereich. Dafür gibt es die anderen Datentypen.

Die Wertebereiche der Standard-Datentypen in C sind **plattformspezifisch**, hängen also davon ab, für welchen Rechner bzw. welche **Plattform** ein Programm kompiliert wird. Nehmen wir mal an, dass ein `short` 2 Byte und ein `int` 4 Byte belegt. Dann könnte der folgende Code

```
char ch;
short s;
int i;
```

zu dieser Belegung im Speicher führen:



Wie wir sehen belegt s nun zwei Speicherzellen und i vier. Erstreckt sich eine Variable über mehrere Zellen, bezeichnet man mit der **Speicheradresse der Variablen** immer die Adresse der **ersten** Speicherzelle, die sie belegt. Somit befindet sich ch an der Adresse 6, s befindet sich an der Adresse 7 und i befindet sich an der Adresse 9.

Da s zwei Speicherzellen mit insgesamt 16 Bit belegt, kann die Variable 2^{16} bzw. 65536 unterschiedliche Werte annehmen. i belegt vier Speicherzellen und kann damit 2^{32} bzw. 4.294.967.296 unterschiedliche Werte annehmen. Also: Je mehr Speicher eine Variable belegt, desto mehr unterschiedliche Werte kann sie darstellen.

Wie bereits erwähnt, gibt der Datentyp auch an, wie der gespeicherte Binärwert *interpretiert* wird. In C unterscheidet man hier zunächst zwischen Ganzzahlen und Fließkommazahlen.

Zu den Ganzzahlen gehören standardmäßig `char`, `short`, `int` und `long`. Hier gilt: Mehr Speicher bedeutet, dass größere Zahlen gespeichert werden können. Man unterscheidet bei Ganzzahlen, ob sie vorzeichenlos oder vorzeichenbehaftet sind. Eine vorzeichenlose Zahl ist grundsätzlich positiv und wird über das Schlüsselwort `unsigned` deklariert:

```
unsigned char ch = 136;
unsigned short s = 32012;
unsigned int i = 4294967206;
```

Ein vorzeichenloses char kann die Werte 0 bis 255 speichern. Belegt ein Variable auf einem Rechner 2 Byte (wie die short-Variable im obigen Beispiel), kann sie die Werte 0 bis 65536 speichern, bei einer vorzeichenlosen 4-Byte-Ganzzahl liegen die Werte im Bereich von 0 bis 4.294.967.296.

In der Regel möchte man aber auch mit negativen Zahlen rechnen. Damit eine Ganzzahl auch negative Zahlen speichern kann, schreibt man das Schlüsselwort `signed` davor:

```
signed char ch = -19;
signed short s = 521;
signed int i = -21231231;
```

Standardmäßig sind alle Ganzzahlen in C vorzeichenbehaftet. Daher kann man das Schlüsselwort `signed` auch einfach weglassen:

```
char ch = -19;
short s = 521;
int i = -21231231;
```

Nun stellt sich die Frage, wieso man nicht grundsätzlich alle Variablen mit Vorzeichen deklariert. Der Grund ist, dass ein Vorzeichen ein Bit „kostet“. Eine Variable, die sowohl negative als auch positive Zahlen speichern kann, benutzt das erste Bit ihrer belegten Speicherzellen, um zu definieren, ob die Zahl positiv (erstes Bit ist 0) oder negativ (erstes Bit ist 1) ist. Hier ein Beispiel:

signed char	**0**1000000	entspricht der Zahl	**+**64
signed char	**1**1000000	entspricht der Zahl	**–**64
unsigned char	11000000	entspricht der Zahl	+192

Dadurch kann ein char „nur“ die Werte -128 bis +127 speichern. Eine zwei Byte große Ganzzahl kann die Werte von -32.768 bis +32.767 speichern, eine vier Byte große Ganzzahl wiederum die Werte von -2.147.483.648 bis +2.147.483.647. Für negative Zahlen wird das sogenannte Zweierkomplement verwendet, das hier aber nicht weiter erläutert werden soll.

Man muss die Wertebereiche gut im Auge behalten. Bei dieser Deklaration zum Beispiel

```
char ch = -130;
```

speichert die Variable ch keineswegs den Wert -130, denn der gültige Wertebereich wurde nach unten unterschritten. Die binäre Darstellung von -130 und das „Reinzwängen“ in ein einzelnes Byte führt nun dazu, dass tatsächlich die Zahl 126 (Binär 01111110) in der Speicherzelle gespeichert wird.

Wie bereits erwähnt, sind die Größen aller Datentypen abhängig vom Zielsystem. Wenn man wirklich sichergehen will, dass eine Variable eine feste Anzahl von Bytes belegt, sollte man alternative Datentypen verwenden, die die Speichergröße im Namen kodieren:

```
int8_t i;
int16_t j;
int32_t k;
```

i belegt nun auf jeder Plattform 8 Bit (1 Byte), j belegt 16 Bit (2 Byte), k belegt 32 Bit (4 Byte). Solche Datentypen verwendet man allerdings eher selten und wirklich nur dann, wenn man für mehrere Plattformen programmiert.

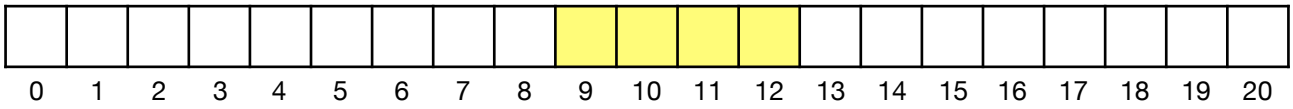
Neben den Ganzzahlen gibt es noch die Fließkommazahlen `float` und `double`. Dabei handelt es sich um Kommazahlen (z.B. 12,5). Der Name „Fließkomma“ stammt daher, dass dieses Zahlenformat in der Binärkodierung die Position des Kommas dynamisch so verschieben kann, dass sowohl sehr kleine als auch sehr große Zahlen möglichst verlustfrei gespeichert werden

können. Ich will auf die konkrete Kodierung an dieser Stelle nicht eingehen. Wichtig anzumerken ist aber, dass der double-Datentyp in der Regel präzisere Zahlen, also in der Regel mehr Nachkommastellen erlaubt. Dafür belegt er auch mehr Speicher – in der Regel doppelt so viel wie der float-Datentyp.

```
float f = 3.5f;
double d = 12.3;
```

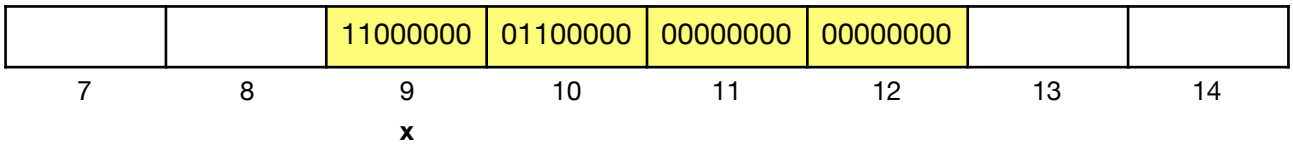
Exkurs: Semantische Lücke

Ich möchte an dieser Stelle nochmal betonen, dass die Auswertung einer Speicherstelle eine reine Interpretationssache ist. Betrachten wir mal das folgende Speicherbild:



Wir sehen hier vier belegte Speicherstellen. Dies könnte eine 4-Byte-Ganzzahl sein oder eine 4 Byte große Fließkommazahl, vielleicht ist es aber auch ein Stück Programmcode, oder eine Rücksprungadresse nach einem Funktionsaufruf, oder etwas Ungültiges … Wir wissen es nicht. In der Speicherstelle steht *irgendetwas*. Eine Speicherstelle verrät uns grundsätzlich nicht, was in ihr gespeichert ist. Dies bezeichnet man im Von-Neumann-Modell als **semantische Lücke**. Nur durch die Interpretation durch ein Programm, erhält der Inhalt von Speicherzellen einen Sinn.

Nehmen wir zum Beispiel an, an den Speicherstellen 9-12 sei eine Variable x mit folgender binärer Belegung gespeichert:



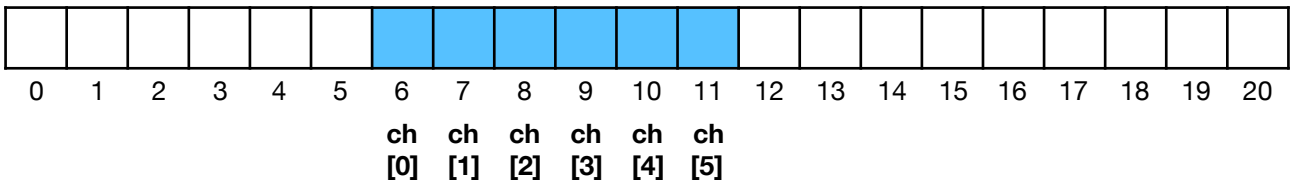
Ist x vom Datentyp float, wird die Binärkodierung als -3,5 interpretriert (32-Bit-IEEE-Float-Format). Ist x hingegen als int deklariert, wird der Inhalt zu -1.067.450.368 aufgelöst (man beachte die führende 1, die ein negatives Vorzeichen anzeigt). Als vorzeichenloses unsigned int ergibt die gleiche Speicherstelle den Wert 3.227.516.928.

Arrays im Speicher

Ein Array ermöglicht es, eine Sammlung von Werten desselben Typs hintereinander in einem Feld zu speichern. Das Array

```
char ch[6];
```

speichert sechs Elemente vom Typ char:

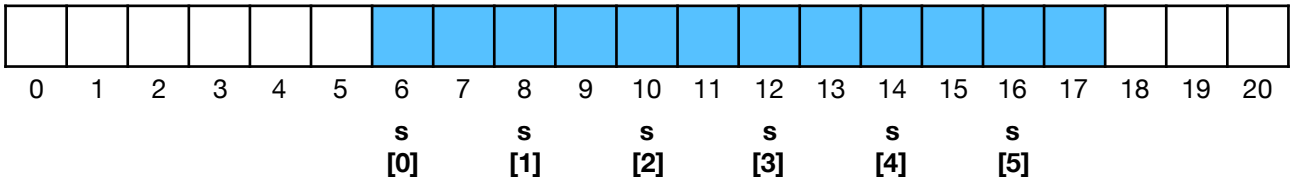


Der Zugriff auf die einzelnen Elemente erfolgt über eckige Klammern, wobei wie üblich darauf zu achten ist, dass das erste Element den Index 0 besitzt. Der Zugriff auf ein Element außerhalb des Arrays führt zu einem Absturz oder zu undefiniertem Verhalten und sollte natürlich vermieden werden.

Wie man sieht, liegen die einzelnen Elemente eines Arrays hintereinander im Speicher. Hier mal ein Beispiel mit short-Werten, die auf unserem fiktiven Rechner 2 Byte belegen:

```
short s[6];
```

führt zu einem solchen Speicherabbild:



Man kann sich grundsätzlich darauf verlassen, dass Array-Elemente hintereinander im Speicher angeordnet sind. Das wird später wichtig, wenn wir über Zeigerarithmetik sprechen.

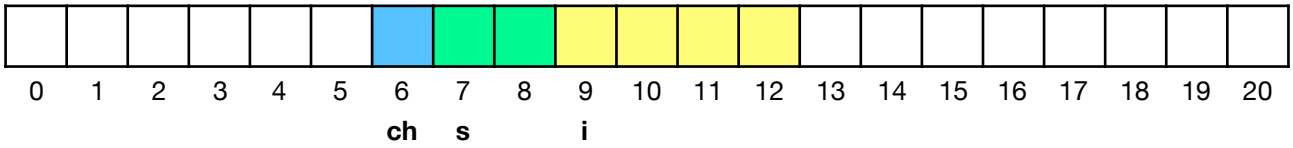
Eine Sonderstellung unter den Arrays besitzen die Strings, die sich dadurch auszeichnen, dass am Ende einer Zeichenkette stets ein 0-Byte (\0) steht, das den String abschließt.

Zeiger

Kommen wir nun zu den Zeigern. Nehmen wir mal an, wir haben drei Variablen deklariert:

```
char ch = 2;
short s = 8;
int i = -15;
```

Die Belegung im Speicher sieht jetzt zum Beispiel so aus:



Wir wissen bereits, dass ch an der Adresse 6, s an der Adresse 7 und i an der Adresse 9 steht. Diese Werte würden wir in C nun gerne ermitteln. Dafür gibt es in den Adress-Operator &, den man einfach vor eine Variable schreibt, um ihre Adresse zu ermitteln. So ergibt der Ausdruck

```
&s
```

den Wert 7. Der Ausdruck

```
&i
```

ergibt den Wert 9. Wollen wir einen solchen Wert speichern, benötigen wir einen **Zeiger**.

Zeiger sind Variablen, die Speicheradressen speichern. Manchmal werden sie auch mit ihrem englischen Begriff als „Pointer“ bezeichnet. Sie sind in der Regel typisiert, d.h. sie besitzen einen Typ, in der Regel den Datentyp der Variablen, auf den sie zeigen. Hier ist zum Beispiel ein Zeiger p („pointer“), der auf die Variable i zeigt.

```
int* p = &i;
```

p ist nun ein auf int typisierter Zeiger, der die Speicheradresse von i, d.h. den Wert 9, speichert. Wie wir oben gelernt haben, kann man einer oder mehrerer Speicherzellen nicht ansehen, was ihr Inhalt bedeutet (semantische Lücke). Deshalb typisiert man Zeiger. Durch das int* wissen wir, dass der Zeiger auf einen Wert zeigt, der (auf unserem Rechner) 4 Byte belegt und eine vorzeichenbehaftete Ganzzahl enthält.

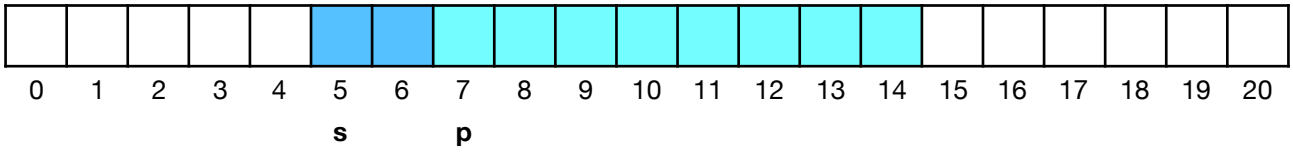
Ein Zeiger belegt wie jede andere Variable Speicher, und zwar genau so viel, dass er *jede* Speicheradresse im Arbeitsspeicher enthalten kann. Daraus folgt, dass jeder Zeiger – unabhängig von seiner Typisierung – gleich viel Speicher belegt, so wie im folgenden Beispiel die Zeiger pi und si:

```
int* pi = &i;
int* si = &s;
```

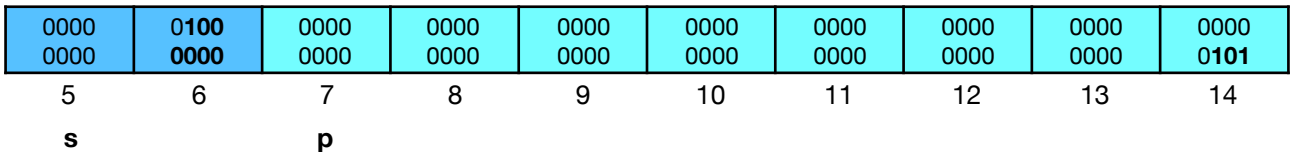
Würde unserer Arbeitsspeicher nur 256 Zellen enthalten, könnte man einen Zeiger in einem Byte speichern. In heutigen 64-Bit-Betriebssystem verbraucht ein einzelner Zeiger in der Regel ganze 8 Byte, um jede Stelle im Speicher adressieren zu können. Schauen wir uns mal ein Beispiel in einer 64-Bit-Architektur an:

```
short s = 64;
short* p = &s;
```

Der Code könnte zu diesem Speicherbild führen:



Während die Variable ch also nur zwei Byte belegt, belegt der Zeiger danach ganze acht Byte. „Zoomen“ wir mal etwas ran und schauen ins die konkreten Inhalte der Speicherzellen an.



s speichert also den Wert 64 (Binär 1000000) und p speichert den Wert 5 (Binär 101), die Adresse von s.

Dereferenzierung

Das Speichern von Adressen durch Zeiger ist natürlich nur sinnvoll, wenn wir uns auch die Inhalte der Adressen anschauen können. Mit dem **Inhaltsoperator** * können wir nachschauen, was sich an der Speicheradresse, die ein Zeiger referenziert, befindet. Man bezeichnet diesen Vorgang als **Dereferenzierung**, weil man die Referenz auf eine Variable über seine Adresse wieder auflöst und den Inhalt ausliest.

Ein Beispiel:

```
short s = 64;
short* p = &s;
short s2 = *p;
```

In der letzten Zeile wird der short-Zeiger p über *p dereferenziert, d.h. der Ausdruck *p wird bei der Ausführung durch den Inhalt der Speicherzelle ersetzt, auf den p zeigt:

0000 0000	0**100 0000**	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0**101**	0000 0000	0**100 0000**
5	6	7	8	9	10	11	12	13	14	15	16
s		**p**								**s2**	

In der Variable s2 steht nun also der Inhalt, der sich hinter p verbirgt.

Bei der Dereferenzierung wird der Inhalt der Speicheradresse **interpretriert**, daher muss der Zeiger zwingend typisiert sein. Im obigen Beispiel arbeiten wir mit einem short-Zeiger. Beim Inhaltsoperator weiß der Compiler also, dass an der Speicheradresse von p eine (in diesem System) 2 Byte große, vorzeichenbehaftete Ganzzahl gespeichert ist. Ein anderer Typ führt zu einer anderen Interpretation. Ändern wir das Beispiel mal leicht ab:

```
short s = 64;
char* p = (char*) &s;
char ch2 = *p;
```

Wie zuvor haben wir hier eine short-Variable s deklariert und den Wert 64 zugewiesen. In der nächsten Zeile haben wir die Adresse von s mittels &s ermittelt, aber dieses Mal auf einen **char**-Zeiger konvertiert. In der letzten Zeile dereferenzieren wir nun den char-Zeiger und speichern das Ergebnis in ch2. Obwohl p dem gleichen Wert wie zuvor – die Speicheradresse von s – enthält, wird bei der Dereferenzierung ein anderer Wert ermittelt, nämlich eine 1-Byte große, vorzeichenbehaftete Ganzzahl. Konkret wird also die erste Speicherzelle von s ausgelesen und in ein char gespeichert. Der Wert von ch2 ist demnach 0:

0000 0000	0**100** 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 **1010**	0000 0000	
5	6	7	8	9	10	11	12	13	14	15	16
s		**p**								**ch2**	

Die Dereferenzierung einer Speicheradresse führt also zu einer Interpretation der Speicherstelle und hängt vom Typ des Zeigers ab.

Untypisierte Zeiger

Soll ein Zeiger nur eine Speicheradresse speichern, ohne dass diese dereferenziert wird, kann man einen untypisierten Zeiger verwenden. Diese besitzen den Typ void*.

```
short s = 64;
void* p = (void*) &s;
```

Ein untypisierter Zeiger dient lediglich dem „Transport“ von Speicheradressen, also der reinen Speicherung einer Adresse, um den Zeiger später wieder in einen typisierten Zeiger umzuwandeln.

Eine Anwendung dafür sind die Funktionen qsort und bsearch aus der Standardbibliothek. Diese wollen lediglich die Speicheradresse eines Arrays wissen. Die vom Entwickler programmierte Vergleichsfunktion wandelt untypisierte Zeiger schließlich um, um zwei Elemente inhaltlich zu vergleichen.

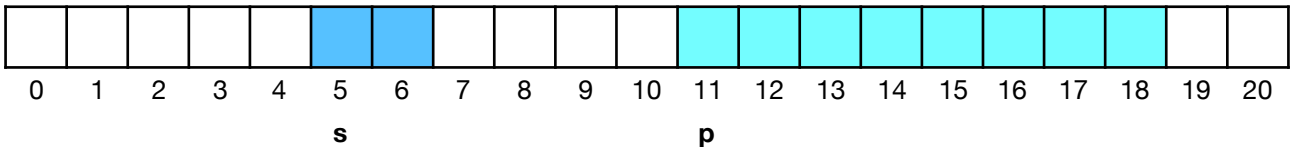
Zeigerarithmetik

Ist ein Zeiger typisiert (also kein void*-Zeiger), kann man mit ihm rechnen. Auf typisierten Zeigern sind die Addition und die Subtraktion definiert.

Wird eine Zahl n auf einen typisieren Zeiger **addiert**, wird die Speicheradresse um n mal der Größe des Zeiger-Datentyps erhöht. Nehmen wir also mal diesen Code:

```
short s = 64;
short* p = &s;
```

mit dem Speicherabbild



p speichert die Adresse von s, also 5. Erhöhen wir jetzt p um eins

```
p++;
```

speichert p den Wert 7. p wurde also im die Größe einer short-Variablen – in diesem Fall 2 Byte – erhöht. Nach einer *weiteren* Erhöhung um 2

```
p += 2;
```

zeigt p auf die Speicherstelle 11. Dies ist äquivalent dazu, als hätte man

```
short* p = &s + 3;
```

geschrieben, was der Rechnung 5 + 3 · sizeof(short) also 5 + 6 entspricht.

Die Speicherarithmetik ist besonders dann interessant, wenn man mit Arrays arbeitet. Nehmen wir mal an, wir haben ein Array wie folgt definiert:

```
short array[1000];
```

Wir möchten nun auf das 101. Element zugreifen. Das geht natürlich über den Ausdruck:

```
array[100]
```

Alternativ kann man ein Zeiger auf das erste Element benutzen und 100 drauf addieren.

```
short* p = &array[0] + 100;
```

Da Arrays Zeiger auf Speicherbereiche sind, kann man statt &array[0] auch einfach array schreiben:

```
short* p = array + 100;
```

Wenn man p nun dereferenziert (*p), erhält man den Inhalt des 101. Elements. Für einen einzelnen Array-Zugriff ist die Speicherarithmetik nicht interessant. Wenn wir aber ein ganzes Array durchlaufen wollen, ist es schneller, mit einem Zeiger auf das erste Element zu starten und diesen in jeder Iteration um eins zu verschieben:

```
short* p = array;
for(int i = 0; i < 1000; i++, p++) {
    short value = *p;
    // mache etwas mit value oder direkt mit *p
}
```

In jeder Iteration der Schleife kommen wir auf den Inhalt des aktuellen Elements über *p. Das ist schneller, als array[i] zu schreiben, da array[i] vom Compiler in den Ausdruck *(array + i) umgewandelt wird. Es müsste also jedes Mal die Startadresse des Arrays ermittelt und um i erhöht werden. p++ ist die schnellere Operation.

Bei Strings ist die Verwendung von Speicherarithmetik besonders elegant, wie diese Schleife zeigt, die einen String in Großbuchstaben umwandelt:

```
char str[64];
strcpy(str, "Bob, the builder");

for(char* p = str; *p != 0; p++) {
    *p = toupper(*p);
}
```

Die Schleife wird über einen Zeiger auf das erste Element des Strings gestartet. Wie gesagt: str und &str[0] können synonym verwendet werden. Die Schleife prüft nun jedes Mal über *p != 0, ob das Ende der Schleife erreicht ist. Ist dies nicht der Fall, wird der aktuelle Buchstabe in einen

Großbuchstaben umgewandelt. Anschließend wandern wir zum nächsten Zeichen über p++ und prüfen die Schleifenbedingung erneut, solange bis das abschließende 0-Byte erreicht ist.

Neben der Addition ist auch eine **Subtraktion auf Zeigern** definiert. Die Subtraktion von zwei Zeigern ergibt die *Anzahl der Elemente* zwischen zwei Zeigern. Sie ist naheliegender Weise zur Addition invers. Definiert man also zwei Zeiger p und q wie folgt

```
short array[1000];
short* p = array;
short* q = p + 3;
```

ergibt der Ausdruck

```
q - p
```

den Wert 3.

Man kann die Subtraktion auch anders auffassen: Es ist die Anzahl der Bytes zwischen den Speicheradressen, dividiert durch die Größe des Datentyps. Beispiel: Nehmen wir an, dass das obige Feld array an der Speicheradresse 10 beginnt und ein short 2 Byte groß ist. Dann gilt p = 10 und q = 16[1]. Der Ausdruck q – p ergibt dann (16 - 10) / sizeof(short), also (16 - 10) / 2 = 3.

Auf untypisierten Zeigern ist Zeigerarithmetik nicht sinnvoll, da die Größe des referenzierten Typs nicht bekannt ist. Also: Keine Addition oder Subtraktion auf void*-Zeigern. Wie oben angemerkt, dienen untypisierte Zeiger ausschließlich dem Transport von Speicheradressen.

Speicherverletzungen und 0-Zeiger

Die Dereferenzierung eines Zeigers, der auf Speicher zeigt, der nicht zu Ihrem Programm gehört, führt zu undefiniertem Verhalten. Im besten Fall stürzt Ihr Programm ab (dann wissen Sie über den Debugger direkt, wo Sie etwas falsch gemacht haben). Im schlimmsten Fall macht Ihr Programm zu einem späterem Zeitpunkt irgendetwas merkwürdiges, weil das Speicherbild beschädigt wurde, so dass die Fehlersuche hier sehr schwer wird.

So etwas passiert relativ leicht, z.B. in diesen Fällen:

- Ihre Funktion gibt einen Zeiger auf eine lokale Variable zurück. Diese wird nach Ende der Funktion aber vom Stack gelöscht, wodurch der Zeiger ungültig wird.
- Sie verwenden einen Zeiger, den Sie noch nicht initialisiert haben.
- Sie lassen einen Zeiger über die Grenzen eines Arrays laufen.
- Sie verwenden einen Zeiger auf dynamisch reservierten Speicher weiter, obwohl Sie ihn zuvor mit der free-Funktion freigegeben haben.

Der Großteil der Abstürze in C-Programmen ist auf Speicherzugriffsfehler zurückzuführen.

Wir haben grundsätzlich keine Möglichkeit festzustellen, ob ein Zeiger auf einen gültigen Speicherbereich zeigt (da ist sie wieder, die semantische Lücke). Um aber bewusst zu kommunizieren, dass ein Zeiger gerade auf „nichts“ zeigt, setzen wir ihn auf 0 oder NULL:

---

[1] q = p + 3 · sizeof(short) = 10 + 3 · 2 = 16

```
free(p);
p = NULL;
```

Bevor man nun mit dem Zeiger arbeitet, kann man sicherheitshalber erst prüfen, ob er NULL ist, bevor man ihn dereferenziert.

Anwendung von Zeigern

Es gibt zwei hauptsächliche Anwendungsgebiete für Zeiger:

1. Die **dynamische Speicherverwaltung**. Wird ein Speicher auf dem Heap reserviert (calloc, malloc, realloc), ist der Rückgabewert grundsätzlich ein Zeiger.
2. Die **schnelle Navigation** durch Arrays, wie im Abschnitt über Zeigerarithmetik erklärt.

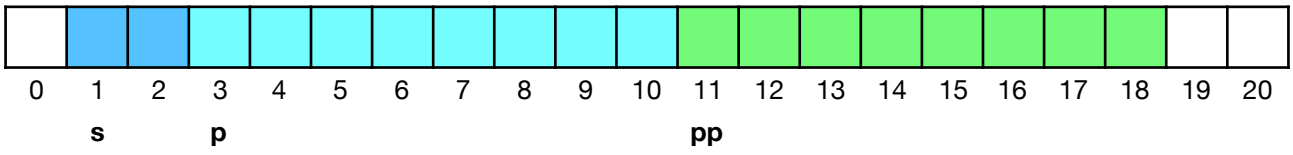
Eine seltenere, aber durchaus wichtige Anwendung ist der **Zugriff auf explizite Speicheradressen** oder Hardware-Register. Wenn Sie ein Programm für ein eingebettetes System, z.B. für einen Steuer-Chip eines medizinischen Geräts schreiben, kann es sein, dass ein Sensorwert, beispielsweise der Blutzucker eines Patienten, an einer fest verdrahteten Speicheradresse steht. Diese könnten Sie nur über einen Zeiger auslesen, dem Sie einen festen Wert zuweisen.

Zeiger auf Zeiger

Zeiger sind Variablen. Daher kann ein Zeiger auch die Speicheradresse eines anderen Zeigers enthalten. Man nennt dies einen Zeiger auf einen Zeiger. Bei der Deklaration wird ein zweiter Stern angegeben:

```
short s = 1000;
short *p = &s;
short **pp = &p;
```

s ist nun eine short-Variable, p ein Zeiger auf s, und pp ein Zeiger auf p. Im Speicher könnte das so aussehen:



p enthält den Wert 1, die Adresse von s. pp enthält den Wert 3, die Adresse von p.

Um von pp auf den Wert von s zu kommen, sind zwei Dereferenzierungen notwendig:

```
**pp
```

Der Ausdruck **pp ergibt den ursprünglichen Wert von s, 1000. Der Ausdruck *pp ergibt wiederum den Wert, den pp referenziert, also den Zeiger p. *pp entspricht also p, d.h. dem Wert 1.

Zeiger auf Zeiger werden in der Regel verwendet, wenn ein Zeiger als *Call-by-reference* übergeben wird. Also dann, wenn eine Funktion einen übergebenen Zeiger ändern soll, so dass er auch für den Aufrufer geändert ist. Die folgende Methode

```
void reserveArray(int** array, size_t size) {
   *array = malloc(sizeof(int) * size);
}
```

reserviert ein int-Array mit size Elementen und speichert die Speicheradresse im Parameter array. Wird die Methode nun mit einer Referenz auf einen Zeiger (Call-by-Reference) aufgerufen

```
int* numbers;
reserveArray(&numbers, 10);
```

ist in numbers *nach* dem Aufruf die Startadresse des reservierten Speichers gespeichert, oder NULL, falls die Reservierung fehlgeschlagen ist.

Grundsätzlich sind auch komplexere Konstruktionen denkbar, z.B. Zeiger auf Zeiger auf Zeiger. Diese kommen in der Praxis aber selten vor und sind unhandlich, so dass man eher auf andere Datenstrukturen zugreift.