8 Einführung in Pointer und Arrays

Pointer sind ein elementarer Bestandteil der Programmiersprache C. Um mit C praxistauglich programmieren zu können, sollte das Konzept der Pointer verstanden sein. Dazu dient diese Einführung in Pointer und Arrays. Eine Fortsetzung dieses Themas finden Sie in Kapitel → 12.

8.1 Pointer-Typen und Pointer-Variablen

Der Arbeitsspeicher eines Rechners ist in Speicherzellen eingeteilt. Jede Speicherzelle trägt eine Nummer.

> Die Nummer einer Speicherzelle wird als **Adresse** bezeichnet.

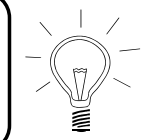
Ist der Speicher eines Rechners byteweise (in der Regel 1 Byte = 8 Bits) aufgebaut, so sagt man, er sei byteweise adressierbar. Über Adressen sind dann einzelne Bytes ansprechbar.

> Einzelne Bits können nicht adressiert werden.

Eine Variable belegt ein oder mehrere Bytes im Speicher. Je nach Typ der Variablen benötigt sie mehr oder weniger Bytes. Eine `float`-Variable beispielsweise benötigt 4 Bytes.

Im Arbeitsspeicher werden somit für eine `float`-Variable 4 Bytes reserviert. Das erste Byte dient dabei als Anfangspunkt. Die Adresse dieses ersten Bytes ist eine eindeutige Position, mit welcher die Variable lokalisiert werden kann. Mithilfe eines **Pointers** kann diese Adresse gespeichert werden.

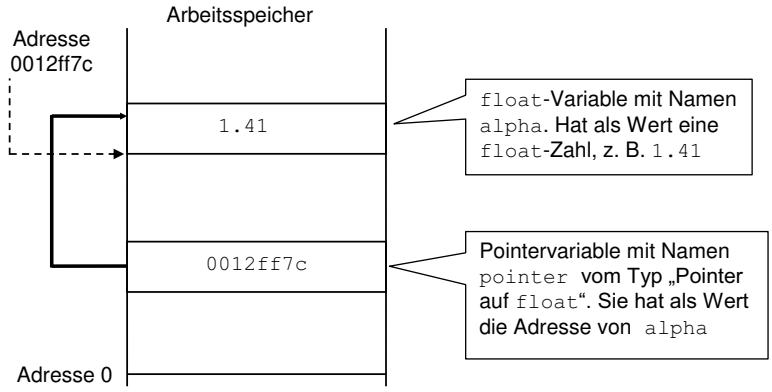
> Ein Pointer (**Zeiger**) ist eine Variable, welche die Adresse einer im Speicher sich befindlichen Variablen aufnehmen kann.



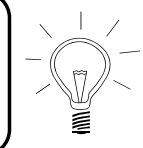
Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_8.

J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_8

Im folgenden Bild wird eine Pointer-Variable auf ein Speicherobjekt schematisch aufgezeigt:



Pointer und Speicherobjekte sind vom Typ her gekoppelt. Ist das Objekt vom Typ `X`, so braucht man einen Pointer vom Typ „Pointer auf `X`“, um auf dieses Objekt zeigen zu können.



Es ist also beispielsweise nicht erlaubt, einen Pointer auf `int` auf eine `double`-Variable zeigen zu lassen. Daher ist es nötig, den Datentyp anzugeben, den der Pointer referenzieren soll – das heißt auf den er zeigen beziehungsweise verweisen soll.

8.1.1 Definition von Pointer-Variablen

Ein Pointer wird formal wie eine Variable definiert, nur dass mittels eines Sternchens festgelegt wird, dass es sich um einen Pointer handelt. Die allgemeine Form der Definition eines Pointers ist:

```
Typname* Pointername;
```

`Typname*` ist der Datentyp des Pointers
`Pointername` ist der Name des Pointers.

Dabei ist `Pointername` ein Pointer auf den Datentyp `Typname`. Diese Definition wird von rechts nach links gelesen, wobei man den `*` als „ist Pointer auf" liest: `Pointername` ist ein Pointer auf `Typname`.

Durch diese Definition wird eine Variable `Pointername` vom Typ „Pointer auf `Typname`" definiert. Konkrete Beispiele für die Definition von Pointer-Variablen sind:

```
int*   pointer1;
float* pointer2;
```

`pointer1` ist ein Pointer auf `int`, `pointer2` ist ein Pointer auf `float`. Der Datentyp dieser Pointer-Variablen ist `int*` beziehungsweise `float*`. Durch die Vereinbarung sind Pointer und zugeordneter Typ miteinander verbunden.

Es ist wichtig zu verstehen, dass durch die Definition einer Pointer-Variablen nur Speicherplatz für die Pointer-Variable selbst reserviert wird, jedoch nicht für das Objekt, auf welches der Pointer schlussendlich zeigen wird.

Da Variablen prinzipiell an jeder beliebigen Stelle des Adressraumes (mit Ausnahme der Adresse 0) liegen können, müssen mit einem Pointer letztendlich alle Adressen dargestellt werden können. Daher werden zur Speicherung eines Pointers genauso viele Bytes verwendet, wie es die interne Darstellung einer Adresse erfordert. Je nach Prozessorarchitektur kann eine Adresse somit unterschiedlich breit sein. Lange Zeit waren 32-Bit-Adressen üblich, mittlerweile setzen sich 64-Bit-Architekturen durch. Eine Adresse belegt auf modernen Maschinen somit normalerweise 4 oder 8 Bytes.

8.1.2 Wo gehört das Sternchen hin?

An dieser Stelle sei anzumerken, dass das Pointer-Zeichen (das Sternchen) formal zur Variablen und nicht zum Typ zugehörig ist.

```
int *pointer1;   // Wie es der Compiler liest.
int* pointer1;   // Wie man es semantisch versteht.
```

Der Compiler liest somit „`pointer1` ist eine Pointer-Variable mit dem zu referenzierenden Typ `int`“. Intuitiv liest sich der Code jedoch „`pointer1` ist vom Typ `int`-Pointer“.

Auch gibt es Quellen, welche das Sternchen in die Mitte platzieren, um es optisch hervorzuheben:

```
int * pointer1;
```

Syntaktisch (sprich im Quellcode geschrieben) spielt es keine Rolle, wo das Sternchen steht. Die semantische Unterscheidung ist jedoch wichtig, wenn mehrere Pointer-Variablen in einer Zeile deklariert werden (siehe dazu Kapitel → 7.4.1). Beachten Sie die folgenden Unterschiede bei der Definition von Pointer-Variablen:

```
int *pointer1, pointer2;
int *pointer3, *pointer4;
```

In der ersten Zeile werden eine Pointer-Variable und eine int-Variable definiert, in der zweiten Zeile dagegen zwei Pointer-Variablen.

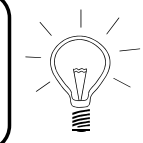
Um Fehldeklarationen zu vermeiden, sollte eine Pointer-Variable immer in einer separaten Zeile definiert werden.

Hier in diesem Buch wird das Sternchen in den Code-Beispielen grundsätzlich zum Typ geschrieben.

8.1.3 Adress-Operator

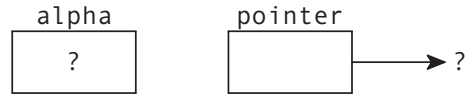
Die einfachste Möglichkeit, einen Pointer auf ein Objekt zeigen zu lassen, besteht darin, den **Adress-Operator** & auf eine Variable anzuwenden, denn es gilt:

> Ist `alpha` eine Variable vom Typ `X`, so liefert der Ausdruck `&alpha` einen Pointer auf das Objekt `alpha` vom Typ „Pointer auf `X`“.



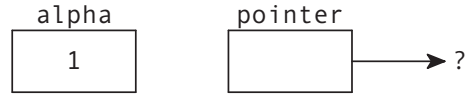
Im folgenden Beispiel wird eine `int`-Variable `alpha` definiert und ein Pointer `pointer` auf eine `int`-Variable:

```
int alpha;
int* pointer;
```



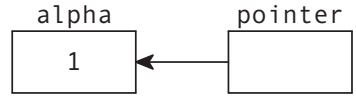
Durch die folgende Zuweisung `alpha = 1` wird der `int`-Variablen `alpha` der Wert 1 zugewiesen, das heißt an der Stelle des Adressraums des Rechners, an der `alpha` gespeichert wird, befindet sich nun der Wert 1. Zu diesem Zeitpunkt ist der Wert des Pointers noch undefiniert, denn ihm wurde noch nichts zugewiesen:

```
alpha = 1;
```

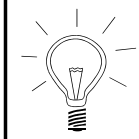


Erst durch die Zuweisung `pointer = &alpha` erhält der Pointer `pointer` einen definierten Wert, nämlich die Adresse der Variablen `alpha`. Dies zeigt das folgende Bild:

```
pointer = &alpha;
```



> Der Adress-Operator liefert als Ergebnis die Adresse einer bereits im Speicher angelegten Variablen und diese Adresse kann in einer Pointer-Variablen (oder: in einem Pointer) gespeichert werden.

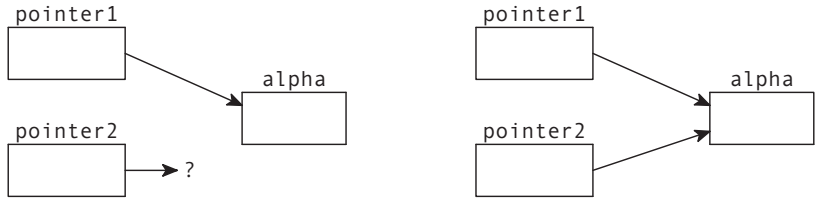


Eine Ausnahme bilden hierbei die `register`-Variablen (siehe Kapitel → 15.6). Da sie gegebenenfalls vom Computer in Registern des Prozessors abgelegt werden, kann der Adress-Operator hier nicht verwendet werden.

Mittels des Adress-Operators kann man einem Pointer also grundsätzlich jede beliebige Adresse zuweisen. Da Adressen nichts anderes als Integer sind, könnte man versucht sein, Adressen direkt als solche hinzuschreiben. Dies ist jedoch nicht erlaubt. Erlaubt sind jedoch Zuweisungen von anderen Pointern, sofern sie denselben Typ besitzen:

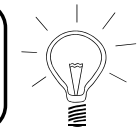
```
int* pointer1 = &alpha;
int* pointer2;
pointer2 = pointer1;
```

Dadurch wird dem Pointer `pointer2` der Wert des Pointers `pointer1` zugewiesen. Nach der Zuweisung haben beide Pointer denselben Inhalt und zeigen damit beide auf das Objekt, auf das zunächst von `pointer1` verwiesen wurde:



Um zu verifizieren, dass beide Pointer auf dieselbe Adresse zeigen, kann man sich die Adresse auch ausgeben lassen.

Pointer können mittels `printf()` mit dem Formatelement `%p` ausgegeben werden.



```
printf("alpha liegt an der Adresse %p\n", pointer1);
printf("alpha liegt an der Adresse %p\n", pointer2);
```

Je nach Compiler und Laufzeitumgebung kann die Ausgabe unterschiedlich ausfallen. Eine mögliche Ausgabe wäre:

```
alpha liegt an der Adresse 020f35b0
alpha liegt an der Adresse 020f35b0
```

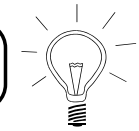
8.1.4 `NULL`-Pointer

Wie im vorherigen Kapitel gezeigt, ist der Wert einer lokalen Pointer-Variablen nach der Variablendefinition zunächst unbestimmt, sprich undefiniert. Der Pointer referenziert also irgendeine Speicherstelle im Adressraum des Programms.

> Ein Pointer als lokale Variable, der nicht initialisiert wurde, enthält einen zufälligen Wert und zeigt somit auf eine beliebige Speicherstelle. Er ist als solches nicht zu unterscheiden von einem Pointer mit einem gültigen Wert.

Um eine Pointer-Variable somit mit einem gültigen Wert zu initialisieren, muss ihr entweder direkt ein gültiger Wert zugewiesen werden, oder aber ihm wird der Wert `NULL` zugewiesen.

> Ein **`NULL`**-Pointer hat den Wert 0.



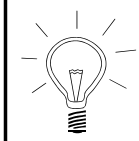
Der Pointer `NULL` ist ein vordefinierter Pointer, dessen Wert sich von allen regulären Pointern unterscheidet. Im C-Standard ist festgelegt, dass Variablen und Funktionen immer an Speicherplätzen abgelegt werden, deren Adressen von 0 verschieden sind.

Die Konstante `NULL` ist in `<stddef.h>` somit als `0` definiert und zeigt damit auf kein gültiges Speicherobjekt. Die kann gleichbedeutend verwendet werden mit einem typfreien Pointer auf die Adresse `0`, sprich `(void*) 0`. Siehe dazu auch Kapitel → 8.2.

```
int* pointer = NULL;
```

Die Zuweisung des Wertes `NULL` zu einem Pointer wird beim Programmieren synonym verwendet zu Bedeutungen wie „leer“, „nicht gesetzt“, „ungültig“ oder „gelöscht“.

Der `NULL`-Pointer wird häufig dazu verwendet, Pointer-Variablen bei der Definition sofort zu initialisieren. Ein Pointer, der mit `NULL` initialisiert ist, zeigt an, dass er noch auf keine gültige Speicherstelle zeigt.



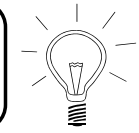
Funktionen, die einen Pointer als Funktionsergebnis liefern, können den `NULL`-Pointer verwenden, um eine erfolglose Aktion anzuzeigen. Liegt kein Fehler vor, so haben sie als Rückgabewert stets die Adresse eines Speicherobjektes, die von 0 verschieden ist.

Ab dem Standard C23 gibt es auch das vorgegebene Schlüsselwort `nullptr`. Damit gleicht sich der C-Standard der Sprache C++ an, wo dieses Schlüsselwort bereis existiert. Die Angabe `nullptr` ist grundsätzlich gleichzustellen mit `NULL`, ist aber zu bevorzugen, da damit durch einen speziell für Nullpointer eingeführten Typ besser auf Typsicherheit geprüft werden kann.

8.1.5 Zugriff auf ein Objekt über einen Pointer

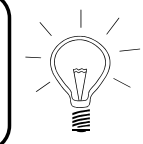
Pointer-Variablen speichern nicht selbst Objekte, sondern zeigen nur darauf. Dies wird auch als „**Referenzieren**“ bezeichnet.

Referenzieren heißt, dass man mit einer Adresse auf ein Speicherobjekt zeigt.



Wurde einem Pointer ein Wert zugewiesen, so will man natürlich auch auf das referenzierte Objekt, also auf das Objekt, auf das der Pointer zeigt, zugreifen können. Dazu gibt es in C den **Dereferenzierungs-Operator** `*`.

Wird der Dereferenzierungs-Operator auf einen Pointer angewandt, so wird direkt auf das Objekt zugegriffen, auf das der Pointer verweist.

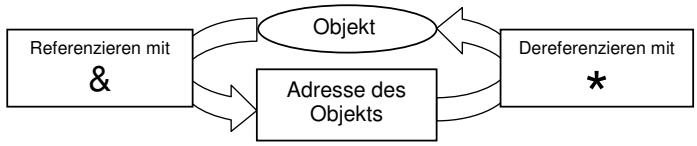


Ein Pointer referenziert ein Objekt – mit anderen Worten, er verweist auf das Objekt. Mit Hilfe des Dereferenzierungs-Operators erhält man aus einem Pointer, also einer Referenz auf ein Objekt, das Objekt selbst.

Beispielsweise wird im Folgenden der Variablen `alpha` der Wert `2` zugewiesen:

```
int alpha = 1;
int* pointer = &alpha;
*pointer = 2;
```

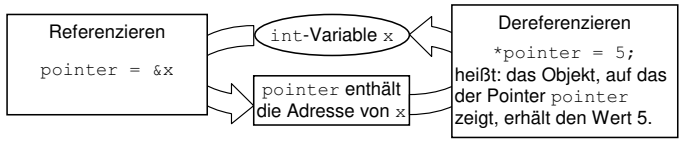
Es ist wichtig zu verstehen, dass der Dereferenzierungs-Operator komplementär zum Adress-Operator sich verhält. Das folgende Bild visualisiert die Verwendung des Adress- und Dereferenzierungs-Operators:



Diese Dualität ist das fundamentale Prinzip der Pointer-Programmierung:

- & wird als Adress-Operator bezeichnet. Mit ihm erhält man die Adresse eines Objekts.
- * wird als Dereferenzierungs-Operator bezeichnet. Mit ihm erhält man den Inhalt des Objekts, auf das der Pointer zeigt.

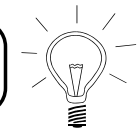
Folgendes ist ein konkretes Beispiel für die Verwendung des Adress- und Dereferenzierungs-Operators:



8.1.6 Beispiele für das Referenzieren und Dereferenzieren

Durch das komplementäre Verhalten des Adress-Operators und des Dereferenzierungs-Operators gilt folgendes:

> `*&alpha` ist äquivalent zu `alpha`.



Folgendes lauffähiges Beispiel veranschaulicht dies:

pointer.c

```
#include <stdio.h>

int main(void) {
  float zahl = 3.5f;
  printf("Adresse von zahl: %p\n", &zahl);
  printf("Wert von zahl: %f\n", *&zahl);
  printf("Wert von zahl: %f\n", *&*&*&*&*&zahl);
  return 0;
}
```

Natürlich ist die wiederholte Anwendung der beiden Operatoren wenig sinnvoll und dient hier nur zur Veranschaulichung der Äquivalenz. Die Ausgabe des Programmes ist die Folgende:

```
Adresse von zahl: 0x16fdff248
Wert von zahl: 3.500000
Wert von zahl: 3.500000
```

Dabei ist zu beachten, dass die Adresse auf einem anderen Rechner natürlich anders ausfallen kann.

Das folgende Programm demonstriert erneut die beiden Operatoren:

ptrOp.c

```
#include <stdio.h>

int main(void) {
  int alpha;
  int* pointer1;
  int* pointer2;

  pointer1 = &alpha;
  *pointer1 = 5;
  printf("%d\n", *pointer1);

  *pointer1 = *pointer1 + 1;
  pointer2 = pointer1;
  printf("%d\n", *pointer2);

  return 0;
}
```

Zuerst wird `pointer1` mithilfe des Adress-Operators die Adresse von `alpha` zugewiesen. Dann wird mittels des Dereferenzierungs-Operators der Wert `5` an die Stelle im Speicher, wohin die Adresse zeigt, zugewiesen und dann ebenfalls mittels des Dereferenzierungs-Operators wiederum ausgelesen und ausgegeben.

Danach wird mittels des Dereferenzierungs-Operators der Wert an der Adresse ausgelesen, mit `1` aufsummiert und erneut zurückgeschrieben. Der zweite Pointer `pointer2` zeigt danach ebenfalls auf die selbe Adresse wie `pointer1` und der Inhalt der Speicherstelle an ebendieser Adresse wird erneut ausgegeben.

Hier die Ausgabe des Programmes:

```
5
6
```

8.1.7 Häufigste Fehlerquellen

Mit Pointern zu arbeiten ist nicht problemlos. Sehr schnell schleichen sich fehlerhafte Adressen in den Verlauf eines Programmes:

```
int* pointer1;
*pointer1 = 6;
```

In diesem Beispiel wurde der Pointer `pointer1` nicht initialisiert, womit er auf eine beliebige Speicherstelle im gesamten Adressraum des Computers zeigt. Damit kann ungewollt eine beliebige Speicherstelle verändert werden. Liegt die Speicherstelle außerhalb des eigenen Adressbereichs, wird das Betriebssystem das Programm abbrechen, um einen Zugriff auf fremde Daten zu unterbinden.

Bestenfalls führt ein Zugriff auf eine ungültige Adresse zum Absturz des Programmes, schlimmstenfalls zu einer sogenannten „**heap corruption**“, was viele Stunden Fehlersuche nach sich ziehen kann.

Der Zugriff auf eine falsche Adresse ist ein häufiger und oftmals schwer aufzufindender Programmierfehler in C.

In neueren Programmiersprachen wie beipspielsweise in Java hat man aus Gründen der Sicherheit keinen Zugriff mehr auf die Adresse einer Variablen im Arbeitsspeicher.

Eng mit dem obigen Beispiel verwandt ist auch der Zugriff auf einen **`NULL`**-Pointer:

```
int* pointer2 = NULL;
*pointer2 = 6;
```

Hier wird versucht, einen Pointer zu dereferenzieren, welcher jedoch die ungültige Adresse `NULL` enthält. In diesem Falle wird das Betriebssystem das Programm abstürzen lassen. Heutzutage spricht man hierbei von einer sogenannten „`NULL`-Pointer-Exception“, abgeleitet von der Ausnahmebehandlung moderner Programmiersprachen.

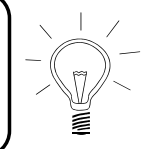
Der Zugriff auf einen `NULL`-Pointer führt zu einem Absturz des Programmes.

Ein `NULL`-Pointer-Zugriff ist einer der häufigsten Absturz-Ursachen von Programmen. Demgegenüber ist jedoch die Fehlersuche bei `NULL`-Pointer-Zugriffen oftmals trivial, im Gegensatz zur obengenannten Heap-Corruption.

8.1.8 Fortgeschrittene Pointer-Definitionen

Bei aufmerksamer Lektüre ist bestimmt aufgefallen, dass eine Pointer-Variable selbst auch ein Speicherobjekt darstellen kann.

Da Pointer-Variablen ebenfalls Speicherobjekte sind, gibt es die Möglichkeit, dass Pointer-Variablen auf Pointer-Variablen zeigen.

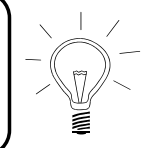


Zudem sei hier angemerkt, dass auch Funktionen Speicherobjekte darstellen und es somit auch möglich ist, Pointer-Variablen auf Funktionen zeigen zu lassen.

Pointer auf Pointer und Pointer auf Funktionen (Funktionspointer) werden in Kapitel → 12 noch behandelt.

8.2 Pointer auf `void`

Wenn bei der Definition des Pointers der Typ der Variablen, auf die der Pointer zeigen soll, noch nicht feststeht, wird ein **Pointer auf den Typ `void`** vereinbart.

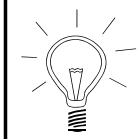


```
void* pointer;
```

Der Pointer auf den Typ `void` darf selbst nicht zum Zugriff auf Objekte verwendet werden, das heißt er darf nicht dereferenziert werden, da nicht bekannt ist, auf welche Objekte er verweist.

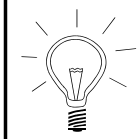
Später kann dann der Pointer in einen Pointer auf einen bestimmten Typ umgewandelt werden.

Der Pointer auf `void` ist ein untypisierter (typfreier, generischer) Pointer. Dieser ist zu allen anderen Pointer-Typen kompatibel und kann insbesondere in Zuweisungen mit typisierten Pointern gemischt werden.



Dies bedeutet, dass in der Programmiersprache C bei einer Zuweisung links des Zuweisungs-Operators ein typfreier Pointer stehen darf und rechts ein typisierter Pointer und auch umgekehrt! Ein Pointer auf `void` umgeht also bei einer Zuweisung die Typüberprüfungen des Compilers. Heutige Compiler geben jedoch auf Wunsch Warnungen aus.

Abgesehen von `void*` darf ohne explizite Typumwandlung kein Pointer auf einen Datentyp an einen Pointer auf einen anderen Datentyp zugewiesen werden. Jeder Pointer auf eine Variable kann durch eine Zuweisung in den Typ `void*` und zurück umgewandelt werden, ohne dass Information verloren geht.



> `void`-Pointer können in jeden anderen Pointer-Typ gecastet werden, solange sie sich nicht in ihren Qualifikatoren (`const`, `restrict`, `volatile`) unterscheiden. Auch der umgekehrte Weg ist möglich: Jeder Pointer-Typ kann in einen entsprechenden `void`-Pointer gecastet werden.

Der Typumwandlungs-Operator – auch cast-Operator genannt – wird in Kapitel → 9.9.5 behandelt. Ein Beispiel zur Nutzung von Pointern auf `void` ist in Kapitel → 12.13 zu finden.

8.3 Arrays

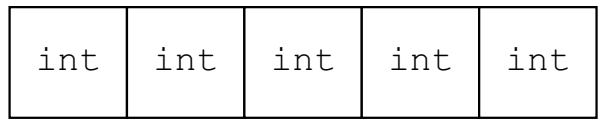
Unter einem **Array** versteht man die geordnete Aneinanderreihung von mehreren Variablen des gleichen Typs unter einem gemeinsamen Namen. Die allgemeine Form der Definition eines eindimensionalen Arrays ist:

```
Typname Arrayname[ANZAHL];
```

Dabei ist `ANZAHL` die Anzahl der **Array-Elemente**. Konkrete Beispiele hierfür sind:

```
int alpha[5];          // Array aus 5 Elementen vom Typ int
char beta[6];          // Array aus 6 Elementen vom Typ char
```

Das folgende Bild visualisiert die Speicherbelegung eines Arrays aus fünf `int`-Elementen:



Alle Elemente des Arrays werden vom Compiler direkt hintereinander im Arbeitsspeicher angelegt.

Ein C-Compiler erkennt ein Array an den eckigen Klammern, die bei der Definition die Anzahl der Elemente enthalten können. Die Anzahl der Elemente muss, wenn vorhanden, immer ein positiver Integer sein. Sie muss normalerweise gegeben sein durch eine Konstante oder einen konstanten Ausdruck. Erst ab dem C11-Standard werden sogenannte VLA (variable length arrays) als Erweiterung erlaubt, bei welchen die Anzahl nicht konstant sein muss.

Ist die Anzahl an Elementen erst zur Laufzeit bekannt, können Arrays mit Hilfe der Funktion `malloc()` dynamisch konstruiert werden. Darüber kann in Kapitel → 18.3 mehr erfahren werden.

8.3.1 Initialisierung eines Arrays

Wird ein Array definiert, so sind die Inhalte dessen zu Beginn unbestimmt. Um ein Array bei der Definition direkt mit Inhalten zu füllen, gibt es die sogenannte „**Aggregats-Initialisierung**“:

```
int alpha[5] = { 32, 62, 17, 105, 30 };
```

Die Zahlen innerhalb der geschweiften Klammern werden vom Compiler automatisch der Reihe nach in das Array geschrieben. Dies funktioniert aber nur bei Initialisierungen, sprich direkt bei der Definition der Array-Variablen.

Da der Compiler die Anzahl der Elemente in einer Aggregats-Liste ermitteln kann, ist es erlaubt, die Anzahl an Elementen bei der Definition wegzulassen:

```
int alpha[] = { 32, 62, 17, 105, 30 };
```

Auch wenn Aggregats-Initialisierungen einfach aussehen, benötigt der Computer Laufzeit, um alle Werte an die richtigen Stellen im Hauptspeicher zu schreiben.

8.3.2 Zugriff auf Elemente eines Arrays

Das folgende Beispiel zeigt ein Array aus fünf `int`-Elementen:

```
int alpha[5];
```

Um nun die einzelnen Elemente des Arrays anzusprechen, kann einfach der Name des Arrays zusammen mit dem **Array-Element-Operator** verwendet werden. Beispielsweise kann so das Element mit Index `3` angesprochen werden:

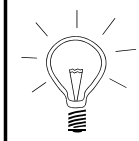
```
alpha[3];
```

Ob dieser Zugriff nun lesend oder schreibend ist, spielt keine Rolle. Folgender Code ist also möglich:

```
alpha[2] = 1234;
int myValue = alpha[2];
```

Hier wurde in der ersten Zeile an die Stelle mit Index `2` der Wert `1234` zugewiesen. In der zweiten Zeile wurde ebendieser Wert wiederum gelesen und einer neuen Variable `myValue` zugewiesen.

> Der Zugriff auf ein Element eines Arrays erfolgt über den **Array-Index**. Hat man ein Array mit n Elementen definiert, so ist darauf zu achten, dass in C die Indizierung der Array-Elemente mit 0 beginnt und bei n - 1 endet.



Da der Index ein Integer sein muss, kann der Index auch aus einer Variablen kommen, wie beispielsweise:

```
int index = 2;
int myValue = alpha[index];
```

C-Compiler erlauben hierbei jeden beliebigen Typ, der zu einem Integer auswertet. Korrekterweise sollte man jedoch für einen Array-Index den speziell hierfür definierten Typ **`size_t`** verwenden:

```
size_t index = 2;
int myValue = alpha[index];
```

Der Sinn eines speziellen Typs für einen Array-Index ist, dass der Typ `int` eine implementationsabhängige Größe hat, welche nicht zwingendermaßen mit der Größe übereinstimmen muss, mit welcher Adressen gebildet werden. In so einem Falle wäre es nicht möglich, mit einem `int` alle Elemente eines sehr großen Arrays zu adressieren. Der Typ `size_t` hingegen erlaubt stets die Adressierung jedes beliebigen Elements.

8.3.3 Arrays und Schleifen

Wir betrachten das folgende, einfache Array:

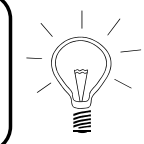
```
#define ANZAHL 5
int alpha[ANZAHL];
```

Um dieses Array mit Werten zu belegen, kann folgendes geschrieben werden:

```
alpha[0] = 1;
alpha[1] = 2;
alpha[2] = 3;
alpha[3] = 4;
alpha[4] = 5;
```

Dies ist jedoch sehr umständlich und birgt großes Fehlerpotential. Viel einfacher ist es, ein Array mittels einer Schleife zu bearbeiten.

> Der Vorteil von Arrays gegenüber mehreren einfachen Variablen ist, dass Arrays sich leicht mit Schleifen bearbeiten lassen.



Diese `for`-Schleife tut dasselbe wie der Code vorher:

```
for (size_t index = 0; index < ANZAHL; index = index + 1) {
  alpha[index] = index + 1;
}
```

Mittels einer Schleife kann man die Werte der Elemente des oben eingeführten Arrays `alpha` auch ausgeben:

```
for (size_t index = 0; index < ANZAHL; index = index + 1) {
  printf("%d, ", alpha[index]);
}
```

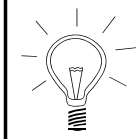
Oder den Durchschnitt aller Werte des Arrays berechnen:

```
double summe = 0.f;
for (size_t index = 0; index < ANZAHL; index = index + 1) {
  summe = summe + alpha[index];
}
printf("Der Durchschnitt ist: %f", summe / ANZAHL);
```

Auf jeden Fall sollte man es sich bei Arrays zur Gewohnheit machen, immer mit symbolischen Konstanten wie beispielsweise `#define ANZAHL 5` und nicht direkt mit literalen Konstanten zu arbeiten. Soll nämlich das vorliegende Beispiel erweitert werden, sodass das Array `10` Elemente enthalten soll anstelle von `5`, so müsste man ohne die Konstante doch an vielen Stellen Änderungen vornehmen, von denen leider gerne welche vergessen werden.

8.3.4 Strings und Arrays

Ein **String** wird vom Compiler intern als ein Array von Zeichen (char-Array) dargestellt. Dabei wird am Schluss ein zusätzliches Zeichen, das Zeichen `'\0'` (**Nullzeichen**) angehängt, um das Stringende zu markieren.



Stringverarbeitungsfunktionen benötigen unbedingt dieses Nullzeichen, damit sie das Stringende erkennen. Deshalb muss bei der Speicherung von Strings stets ein Speicherplatz für das Nullzeichen vorgesehen werden. So hat beispielsweise die folgende Definition nur Platz für 14 Buchstaben und das abschließende `'\0'`-Zeichen:

```
char vorname[15];
```

Werden Strings kopiert oder anderweitig verändert, so muss stets darauf geachtet werden, das `'\0'`-Zeichen ans Ende zu setzen. Das Vergessen dieses Zeichens ist ein häufiger Fehler.

Oftmals wird ein String direkt bei der Definition mit einer String-Konstante initialisiert. Damit man nicht mühsam die Anzahl Zeichen zählen muss, erledigt dies der Compiler mit der folgenden Schreibweise:

```
char stadtName[] = "New York";
```

Hierbei wird auch automatisch das abschließende `'\0'`-Zeichen mitgezählt.

8.3.5 Beispiel für Strings und Arrays

Das folgende Beispiel durchsucht ein char-Array nach dem ersten Zeichen 'a':

charArray.c

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

int main(void) {
  const char eingabe[] = "Dieses Programm findet Buchstaben.";
  size_t index = 0;

  while (eingabe[index] != '\0') {
    if (eingabe[index] == 'a') {
      break;
    }
    index = index + 1;
  }

  if (eingabe[index] == '\0') {
    printf("Ihr String enthaelt kein 'a'\n");
  } else {
    printf("Das erste a befand sich an Index %d.\n", (int)index);
  }

  return 0;
}
```

Die Programmausgabe ist:

```
Das erste a befand sich an Index 12.
```

Das Programm initialisiert das char-Array eingabe mit einem String. Dieser String wird dann in der while-Schleife Zeichen für Zeichen durchlaufen. Die Schleife bricht ab, wenn das '\0'-Zeichen erreicht wird. Jedes Zeichen wird innerhalb der Schleife mit dem gesuchten Zeichen 'a' verglichen. Wird ein 'a' gefunden, dann wird die Schleife mit break verlassen. Am Ende der Schleife wird das Zeichen an der Stelle eingabe[index] erneut mit dem Zeichen '\0' verglichen. Ist das Zeichen gleich '\0', dann wurde der String bis zum Ende durchsucht, ohne ein 'a' zu finden. Ansonsten wurde ein 'a' mit dem Index index gefunden.

Weitere Beispielprogramme mit Arrays und Strings befinden können in Kapitel → 12 nachgelesen werden.

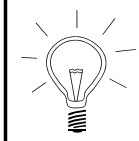
8.4 Der Qualifikator `restrict`

Der **`restrict`**-Qualifikator wird verwendet, wenn zwei oder mehr Arrays oder Pointer lokal definiert sind. Wenn der `restrict`-Qualifikator angegeben wird, so kann der Compiler davon ausgehen, dass sich die beiden Arrays, beziehungsweise die Speicherbereiche, auf welche die Pointer zeigen, nicht überlappen.

Dadurch ist es dem Compiler möglich, bessere **Optimierungen** vorzunehmen. Denn wenn die Speicherbereiche nicht mit `restrict` deklariert werden, muss jede Auswertung einer Array-Variablen oder eines Pointers in der durch das Programm fest vorgegebenen Reihenfolge ausgewertet und zugewiesen werden. Es könnte ja sein, dass in eine Variable ein Wert geschrieben wird, welcher an anderer Stelle wieder gelesen werden muss. In diesem Falle darf der Compiler die beiden Anweisungen nicht austauschen, auch wenn sie rein logisch möglicherweise austauschbar und damit optimierbar wären.

Das Schlüsselwort `restrict` gibt es seit C99.

> Die Verwendung des Schlüsselwortes `restrict` ist insbesondere für die Übergabe von Arrays als Funktionsparameter interessant. Bei der Übergabe eines Arrays wird nicht das gesamte Array, sondern nur ein Pointer auf das erste Element des Arrays übergeben. Mit dem Schlüsselwort `restrict` kann festgelegt werden, dass die Pointer der aktuellen Parameterliste keine Speicherbereiche bezeichnen, die sich überlappen.

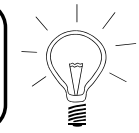


Hier als Beispiel der Prototyp der String-Kopier-Funktion `strcpy()` aus der Standardbibliothek des C11-Standards mit Pointern als Übergabeparameter mit dem Qualifikator `restrict`:

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

Hier bedeutet `restrict` somit konkret, dass `s1` und `s2` auf verschiedene disjunkte Objekte zeigen.

> Hat man in einer Aufrufschnittstelle mehrere `restrict`-Pointer, so müssen sie auf verschiedene Objekte verweisen.



Definiert man einen Pointer `ptr` mit dem Qualifikator `restrict`, so schafft man ein besonderes Verhältnis zwischen diesem Pointer `ptr` und dem referenzierten Objekt: Ein `restrict`-Pointer `ptr` deutet an, dass während der Lebensdauer des entsprechenden Pointers nur mit diesem Pointer `ptr` oder mit einem von diesem Pointer abgeleiteten Pointer, wie beispielsweise `ptr + 1`, auf ein Speicherobjekt wie beispielsweise ein Array von Zeichen zugreifen wird.

Arbeitet man mit einem `restrict`-Pointer, kann ein Compiler Optimierungen des Maschinencodes durchführen, er ist aber dazu nicht verpflichtet.

Bei der Angabe von `restrict` bei einem Pointer auf ein Objekt muss beim Schreiben des Programmes sichergestellt werden, dass der Zugriff auf dieses Speicherobjekt nur über diesen `restrict`-Pointer erfolgt. Die Angabe des Schlüsselworts `restrict` bei der Pointerdefinition ist letztendlich lediglich ein Versprechen für den Compiler, dass ein Zugriff auf das entsprechende Speicherobjekt nur von diesem `restrict`-Pointer aus erfolgt.

Ein Compiler prüft das aber nicht! Überlappen sich Speicherbereiche, welche mit `restrict` deklariert sind, dennoch, so ist das Verhalten des Programms gemäß Standard undefiniert.

Da einige Funktionen der Standardbibliothek wie zum Beispiel die `str-` oder `mem`-Funktionen in Kapitel → 12.9 und → 12.8 Gebrauch von `restrict`-Pointern machen, lohnt es sich bei der Verwendung dieser Funktionen, deren Deklaration genau zu betrachten, um einem „undefinierten Verhalten“ zu entgehen.

8.5 Übungsaufgaben

Aufgabe 1: Pointer und Adress-Operator

Schreiben Sie ein einfaches Programm, das die folgenden Definitionen von Variablen und die geforderten Anweisungen enthält:

- Definition einer Variablen `i` vom Typ `int`
- Definition eines Pointers `ptr` vom Typ `int*`
- Zuweisung der Adresse von `i` an den Pointer `ptr`
- Zuweisung des Wertes 1 an die Variable `i`
- Ausgabe des Wertes des Pointers `ptr`
- Ausgabe des Wertes von `i`
- Ausgabe des Wertes des Objekts, auf das der Pointer `ptr` zeigt, mit Hilfe des Dereferenzierungs-Operators
- Zuweisung des Wertes 2 an das Objekt, auf das der Pointer `ptr` zeigt, mit Hilfe des Dereferenzierungs-Operators
- Ausgabe des Wertes von `i`

Hinweis: Pointer werden bei `printf()` mit dem Formatelement `%p` ausgegeben.

Aufgabe 2: Array-Zugriffe

Überlegen Sie, was das folgende Programm ausgibt. Überzeugen Sie sich durch einen Programmlauf.

arrayExercise.c

```
#include <stdio.h>

int main(void) {
  size_t i;
  int ar[100];

  for (i = 0; i < 100; i = i + 1)
    ar[i] = 1;

  ar[11] = -5;
  ar[12] = ar[12] + 1;
  ar[13] = ar[0] + ar[11] + 4;

  for (i = 10; i <= 14; i = i + 1)
    printf("ar[%2d] = %4d\n", (int)i, ar[i]);

  return 0;
}
```

Aufgabe 3: Eigene Array-Programme

Nutzen Sie das Programm der Aufgabe 8.2 als Vorlage und erstellen Sie je ein eigenes Programm:

a) Definieren Sie einem Array mit 128 Zeichen und weisen Sie diesem die Zeichen des ASCII-Zeichensatzes zu. Geben Sie die Zeichen mit dem ASCII-Code 48 bis 57 am Bildschirm aus.

b) Erstellen Sie ein Array mit 10 `int`-Elementen und initialisieren Sie sie mit beliebigen Werten. Ermitteln Sie, welches Element den größten Wert hat und geben Sie die Nummer des Elements und seinen Wert am Bildschirm aus.

c) Definieren sie zwei Arrays a und b aus je 3 `double`-Elementen. Bestimmen Sie das Skalarprodukt ($a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$) mittels einer Schleife über alle drei Array-Elemente und geben Sie das Resultat am Bildschirm aus.

Aufgabe 4: Fehlende Überprüfung auf Überschreitung der Feldgrenzen bei Arrays

Führen Sie einen Programmlauf mit dem folgenden Programm durch. Analysieren Sie das Ergebnis!

arrayOutOfBounds.c

```
#include <stdio.h>

int main(void) {

  int i = 16;
  int k = 21;
  int l = 22;
  int p = 23;
  int q = 24;

  int ar[100];

  for (size_t i = 0; i < 100; i = i + 1)
    ar[i] = 27;

  printf("i ist %d\n", i);
  printf("ar[-1] ist %d\n", ar[-1]);
  printf("ar[0] ist %d\n", ar[0]);
  printf("ar[100] ist %d\n", ar[100]);
  printf("ar[101] ist %d\n", ar[101]);
  printf("ar[102] ist %d\n", ar[102]);
  printf("ar[103] ist %d\n", ar[103]);
  printf("ar[-2] ist %d\n", ar[-2]);
  printf("ar[-3] ist %d\n", ar[-3]);
  printf("k ist %d\n", k);
  printf("l ist %d\n", l);
  printf("p ist %d\n", p);
  printf("q ist %d\n", q);

  return 0;
}
```

Achtung, lassen Sie sich nicht von diesem Programmierbeispiel zu gewagten Indizierungs- oder Adressierungsmanövern verleiten. Überschreitungen von Feldgrenzen werden heutzutage allgemein als unzulässige Programmierung betrachtet. Das Beispiel dient lediglich der Illustration, was bei einer Überschreitung der Feldgrenzen passieren kann.