12 Fortgeschrittene Programmierung mit Pointern

Pointer und **Arrays** wurden bereits in Kapitel → 8 beschrieben. Pointer sind jedoch in C ein so fundamentaler Bestandteil der Sprache, dass sie ein zusätzliches Kapitel erfordern.

Hier wird auf die fortgeschrittenen Eigenschaften von Pointern und Arrays eingegangen.

12.1 Gleichheit und Unterschiede von Arrays und Pointer

Wie aus Kapitel → 8 bereits bekannt ist, wird ein eindimensionales Array folgendermaßen definiert:

```
int alpha[5];   // Das Array alpha hat Platz fuer 5 int-Zahlen
```

Eine einfache Möglichkeit, einen Pointer auf ein Array-Element zeigen zu lassen, besteht darin, auf der rechten Seite des Zuweisungs-Operators den Adress-Operator & wie folgt auf ein Array-Element anzuwenden:

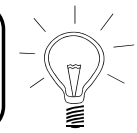
```
int* pointer = &alpha[i];
```

Hat `i` den Wert 1, so zeigt der Pointer `pointer` auf das Array-Element mit dem Index 1.

Dabei gibt es für das erste Element zwei gleichwertige Schreibweisen:

- `alpha[0]`
- `*alpha`

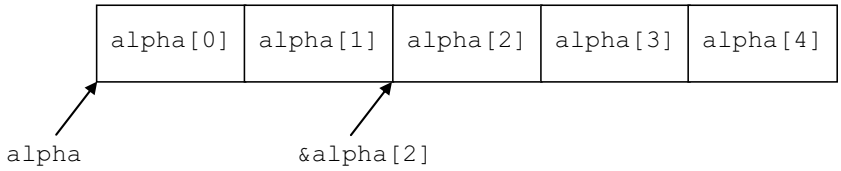
In der Sprache C wird ein **Arrayname** im Allgemeinen zu einem Pointer auf das erste Element des Arrays ausgewertet.



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_12.

© Der/die Autor(en), exklusiv lizenziert an
Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2024
J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_12

Das folgende Bild zeigt verschiedene Pointer auf ein eindimensionales Array:



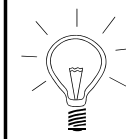
Kapitel → 12.1.1 beschreibt die Ausnahmen, wann dies nicht der Fall ist.

Da ein Arrayname zu einem Pointer ausgewertet wird, ist es in C mit dem Vergleichs-Operator nicht möglich, zwei Arrays auf identischen Inhalt zu überprüfen, wie beispielsweise durch `arr1 == arr2`. Es wird nur verglichen, ob `arr1` und `arr2` auf dieselbe Adresse zeigen.

Für den Vergleich zweier eindimensionaler Arrays gibt es allerdings zwei Möglichkeiten. Die eine Möglichkeit ist die Überprüfung der einzelnen Array-Elemente in einer Schleife. Die andere, elegantere Möglichkeit wird mit der standardisierten Funktion `memcmp()` durchgeführt (siehe Kapitel → 12.8.3). Die Bibliotheksfunktion `memcmp()` führt den byteweisen Vergleich einer Anzahl von Speicherstellen durch, die an über Parameter vorgegebenen Positionen im Adressraum des Speichers liegen.

12.1.1 Arrayname als nicht modifizierbarer L-Wert

Der Name eines Arrays bezeichnet grundsätzlich das (gesamte) Array im Speicher und somit einen L-Wert. Tatsächlich wird dieser L-Wert jedoch nur beim `sizeof`-Operator und dem Adress-Operator & sowie der Initialisierung mittels einer String-Konstanten so angesprochen. Bei allen anderen Operationen wird der Arrayname als die Adresse des ersten Elements ausgewertet, also einem R-Wert. Der Typ des Wertes entspricht einem Pointer auf den Komponenten-Typ des Arrays.



So ist es also möglich, einer entsprechenden Pointer-Variablen direkt die Adresse des ersten Elements zuzuweisen:

```
int alpha[5];
int* pointer = alpha;  // Pointer zeigt auf das erste Array-Element
```

Da es sich bei der Auswertung des Arraynamens um einen R-Wert handelt, sind Ausdrücke wie `alpha++` oder `alpha--` nicht erlaubt, da der Inkrement- und der Dekrement-Operator modifizierbare L-Werte voraussetzen. Ein Arrayname kann auch nicht auf der linken Seite einer Zuweisung stehen, da eine Zuweisung links vom Zuweisungs-Operator ebenfalls einen modifizierbaren L-Wert erfordert.

Einer Pointer-Variablen kann ein Wert zugewiesen werden, einem Arraynamen nicht.

Würde man den Adress-Operator & beim Arraynamen verwenden, so resultiert ein Pointer vom Typ `int (*) [5]`. Dies ist eine etwas verwirrende Typbeschreibung, liest sich jedoch wie ein geklammerter Ausdruck: Zuerst wird die Klammer ausgewertet, sprich, es ist ein Pointer. Dann wird der Rest des Typausdrucks ausgewertet, also ein Array mit 5 `int`-Werten. Zusammengesetzt entspricht dies einem „Pointer auf ein Array mit 5 `int` Elementen“.

Um eine Variable mit diesem Typ zu definieren, muss man also schreiben:

```
int (*pointerToArray)[5] = &alpha;
```

Wenn man nun diese Variable `pointerToArray` mit dem Dereferenzierungs-Operator auswertet, so ergibt sich wiederum das Array. Um nun zusätzlich noch den Pointer auf das erste Element zu erhalten, muss noch ein zusätzlicher Dereferenzierungs-Operator angefügt werden:

```
**pointerToArray = 5;
```

Diese Art der Adressierung und Dereferenzierung bietet insgesamt nicht viele Vorteile beim Programmieren. Sie dient hauptsächlich der Typ-Überprüfung bei der Parameterübergabe. Da das Schreiben der Typausdrücke verwirrend ist, wird entweder auf die Typ-Definition mittels `typedef` zurückgegriffen (siehe Kapitel → 14.2), oder es wird auf die Typ-Überprüfung verzichtet, indem einfach nur der Pointer auf das erste Element übergeben wird.

12.2 Pointerarithmetik

Unter dem Begriff **Pointerarithmetik** fasst man die Menge der zulässigen Operationen mit Pointern zusammen. Folgende Operationen sind erlaubt:

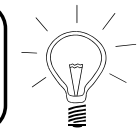
- Zuweisungen mit Pointern
- Addition und Subtraktion bei Pointern
- Vergleiche von Pointern

Bevor die Möglichkeiten der Pointerarithmetik erläutert werden, lohnt sich ein erneutes Studium des in Kapitel → 8.1.4 vorgestellten `NULL`-Pointers.

12.2.1 Zuweisungen mit Pointern

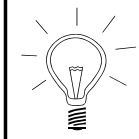
Der Zuweisungs-Operator erlaubt es, Pointer-Variablen eine Adresse zuzuweisen. Die beiden Typen links und rechts des Zuweisungs-Operators sollten jedoch übereinstimmen.

> Pointer unterschiedlicher Datentypen sollten einander nicht zugewiesen werden.



C erlaubt eine Zuweisung, allerdings geben moderne Compiler stets eine Warnung aus. C++ verbietet eine solche Zuweisung, es sei denn, es wird ein expliziter Cast verwendet.

> Pointern vom Typ `void*` dürfen Pointer eines anderen Datentyps zugewiesen werden und Pointern eines beliebigen Datentyps dürfen Pointer vom Typ `void*` zugewiesen werden. Die Typüberprüfung des Compilers wird dabei aufgehoben.



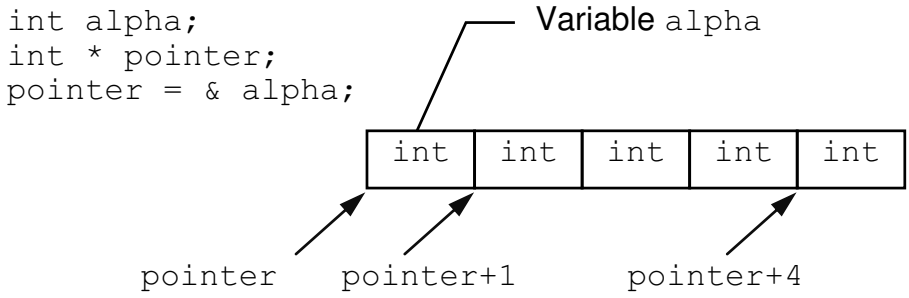
C++ hingegen verbietet die Zuweisung eines **`void`-Pointers** an einen nicht-`void`-Pointer, es sei denn, es wird ein entsprechender Cast verwendet.

Der **`NULL`-Pointer** kann – da er gleichwertig zu `(void*)0` ist – ebenfalls jedem anderen Pointer zugewiesen werden.

12.2.2 Addition von Pointern mit einer Integer-Zahl

Zu einem Pointer kann ein Integer addiert oder von ihm abgezogen werden. Wird ein Pointer vom Typ `int*` um 1 erhöht, so zeigt er um ein `int`-Objekt weiter. Wird ein Pointer vom Typ `float*` um 1 erhöht, so zeigt er um ein `float`-Objekt weiter. Die Erhöhung um 1 bedeutet, dass der Pointer immer um ein Speicherobjekt vom Typ, auf den der Pointer zeigt, weiterläuft.

Das folgende Bild symbolisiert, dass die Pointerarithmetik in Längeneinheiten des Typs, auf den der Pointer zeigt, vonstatten geht:

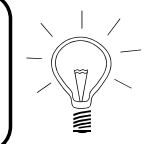


Zeigt ein Pointer auf eine Variable des falschen Typs, so interpretiert der Dereferenzierungs-Operator den Inhalt der Speicherzelle, auf die der Pointer zeigt, gemäß dem Typ des Pointers und nicht gemäß dem Typ der Variablen, die an der Speicherstelle abgelegt wurde.

Die Anzahl Bytes, um welche ein Pointer bei der Erhöhung verschoben wird, wird vom Compiler mit dem `sizeof`-Operator berechnet. Der `sizeof`-Operator wird auf den referenzierten Typ angewandt. Im obigen Beispiel ist dies der Typ `int`. Ein Pointer kann aber auch auf einen zusammengesetzten Typ wie eine Struktur oder ein Array zeigen. Dementsprechend größer wird die Byte-Anzahl sein.

Genauso können Pointer erniedrigt werden. Generell gilt:

> Ein Pointer, der auf ein Element in einem eindimensionalen Array zeigt, darf mit einem Integer (positiv wie negativ) addiert werden.



> Zeigt der Pointer nach einer arithmetischen Operation nicht mehr in das Array, dann ist das Resultat eines Speicherzugriffs über diese Adresse undefiniert. Diese schwerwiegende Speicherverletzung wird als „Array-Out-of-Bounds“ bezeichnet.

Es ist durchaus möglich und manchmal auch beabsichtigt, dass ein Pointer auf eine Position außerhalb eines Arrays zeigt. Man darf nur mit dieser Adresse nicht auf den Speicher zugreifen. Das folgende Beispiel zeigt eine typische Art der Schleifenprogrammierung, bei der am Ende der Pointer auf das erste Element nach dem Array zeigt:

ptrAdd.c

```
#include <stdio.h>

int main(void) {
  int alpha[5] = {1, 2, 3, 4, 5};
  int* pointer = alpha;

  int i = 0;
  while (i < 5) {
    printf("%d\n", *pointer);
    ++pointer;
    ++i;
  }

  return 0;
}
```

Damit nimmt hier beispielsweise beim letzten Schleifendurchgang `pointer` den Wert `alpha + 5` an. Aber er wird nicht angesprochen, womit dieser Quellcode ungefährlich ist.

12.2.3 Subtraktion von zwei Pointern

Ist `pointer1` ein Pointer auf das Element mit Index `i` und `pointer2` ein Pointer auf das Element mit Index `j` eines eindimensionalen Arrays `eindimarray`, so gilt:

```
pointer1 == eindimarray + i
pointer2 == eindimarray + j.
```

Dies gilt, da der Name `eindimarray` zu einem Pointer ausgewertet wird, zu welchem sodann ein Integer hinzuaddiert wird.

Mit dieser Überlegung ist es auch möglich, zwei Pointer voneinander abzuziehen:

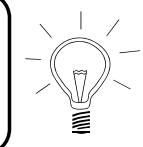
```
pointer2 - pointer1 == (eindimarray + j) - (eindimarray + i).
pointer2 - pointer1 == j - i.
```

Die zweite Zeile zeigt dabei, was passiert, wenn man den Ausdruck rechts kürzt. Übrig bleibt die Rechnung `j - i`, sprich die Anzahl an Elementen zwischen den Elementen mit Index `i` und `j`. Falls `j < i`, ist das Ergebnis negativ.

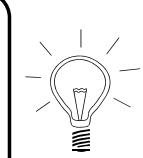
12.2.4 Vergleiche von Pointern

Zwei Pointer können auf Gleichheit beziehungsweise Ungleichheit verglichen werden, wenn beide Pointer denselben Typ haben oder einer der beiden der `NULL`-Pointer ist.

Wenn zwei Pointer auf dasselbe Speicherobjekt zeigen oder beide `NULL` sind, so ergibt der Test auf Gleichheit (==) den booleschen Wert „wahr“.



Für Pointer, die auf Elemente des gleichen eindimensionalen Arrays zeigen, kann aus dem Ergebnis der Vergleiche „größer“ (>) oder „kleiner“ (<) geschlossen werden, dass das eine Element „weiter vorne“ im Array liegt als das andere.

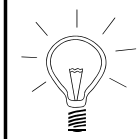


Das gleiche gilt für zwei Pointer, die auf Komponenten derselben Struktur zeigen. In allen anderen Fällen macht ein Vergleich keinen Sinn.

12.3 Initialisierung von Arrays

Die **Initialisierung** von Arrays kann automatisch oder manuell erfolgen.

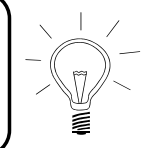
> Globale Arrays werden genauso wie einfache globale Variablen mit 0 initialisiert, das heißt alle Elemente eines globalen Arrays bekommen beim Start des Programmes automatisch den Wert 0 zugewiesen. Lokale Arrays werden nicht automatisch initialisiert.



Wenn der folgende Ausdruck somit im globalen Bereich steht, werden seine Werte automatisch auf 0 initialisiert. Steht er jedoch als eine lokale Definition innerhalb einer Funktion, so sind die Speicherinhalte unbestimmt.

```
int alpha[3];
```

> Um ein Array manuell zu initialisieren, ist nach der eigentlichen Definition des Arrays ein Gleichheitszeichen gefolgt von einer Liste von Initialisierungswerten anzugeben.



Diese **Initialisierungsliste** enthält in geschweiften Klammern {} die einzelnen Werte getrennt durch Kommas. Als Werte können Konstanten oder Ausdrücke aus Konstanten angegeben werden wie im folgenden Beispiel:

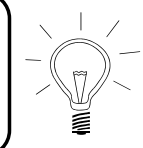
```
int alpha[3] = {1, 2 * 5, 3};
```

Diese Definition ist gleichwertig zu:

```
int alpha[3];
alpha[0] = 1;
alpha[1] = 2 * 5;
alpha[2] = 3;
```

12.3.1 Unvollständige Initialisierung eines Arrays

> Werden bei der Initialisierung von Arrays weniger Werte angegeben als das Array Elemente hat, so werden die restlichen, nicht initialisierten Elemente mit dem Wert 0 belegt.



So werden im Folgenden durch:

```
short alpha[200] = {3, 105, 17};
```

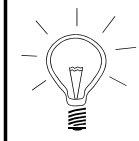
die ersten 3 Array-Elemente explizit mit Werten belegt und die restlichen 197 Elemente haben den Wert 0.

Generell ist es nicht möglich, ein Element in der Mitte eines Arrays zu initialisieren, ohne dass die vorangehenden Elemente auch initialisiert werden.

Enthält die Initialisierungsliste mehr Werte als das Array Elemente hat, so meldet der Compiler einen Fehler.

12.3.2 Initialisierung mit impliziter Längenbestimmung

> Bei der Initialisierung mit impliziter Längenbestimmung wird die Größe des Feldes – also die Anzahl seiner Elemente – nicht bei der Definition angegeben, das heißt die eckigen Klammern bleiben leer. Die Größe wird vom Compiler durch Abzählen der Anzahl Elemente in der Initialisierungsliste festgelegt.



So enthält das folgende Array 4 Elemente:

```
int alpha[] = {1, 2, 3, 4};
```

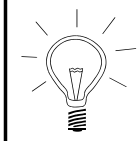
Natürlich hätte man die Größe 4 auch in den eckigen Klammern explizit angeben können.

Das Array `int alpha[]` wird als Array ohne Längenangabe bezeichnet. Die Größe des Arrays ist unbestimmt und stellt somit einen sogenannten „unvollständigen Typ“ dar. Erst durch die Initialisierung wird die Größe des Arrays festgelegt, womit der Typ nicht mehr unvollständig ist.

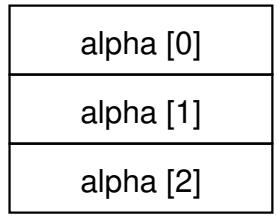
Das Sprachmittel der Initialisierung mit impliziter Längenbestimmung wird vor allem bei Strings verwendet (siehe Kapitel → 12.3.6).

12.3.3 Mehrdimensionale Arrays

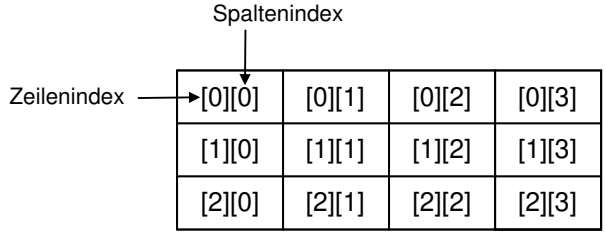
In C ist es wie in anderen Programmiersprachen möglich, **mehrdimensionale Arrays** zu verwenden. Mehrdimensionale Arrays entstehen durch das Anhängen zusätzlicher eckiger Klammern wie beispielsweise `int alpha[3][4];`



Zweidimensionale Arrays lassen sich stets darstellen als ein eindimensionales Array, welches selbst wiederum Arrays in seinen Elementen speichert. Beispielsweise kann `alpha[3][4]` folgendermaßen interpretiert werden:



Man beachte, dass Indizes bei Arrays stets bei 0 zu zählen beginnen. Jedes dieser Elemente des eindimensionalen Arrays ist selbst wiederum ein Array, bestehend aus 4 Elementen:



Damit kann man `alpha` als ein zweidimensionales Array aus 3 Zeilen und 4 Spalten interpretieren. Um auf die einzelnen Elemente zuzugreifen, kann das zweidimensionale Array mit `arrayname[Zeilenindex][Spaltenindex]` angesprochen werden:

```
alpha[1][3]
```

Dieses Element kann auch beim Zuweisungs-Operator auf der linken Seite verwendet werden:

```
alpha[1][3] = 6;
```

Wie ein eindimensionales Array kann auch ein mehrdimensionales Array bereits bei seiner Definition initialisiert werden, beispielsweise durch:

```
int alpha[3][4] = {
  { 1, 3, 5, 7 },
  { 2, 4, 6, 8 },
  { 3, 5, 7, 9 },
};
```

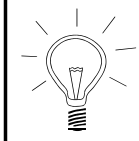
Dabei wird durch `1,3,5,7` die erste Zeile, durch `2,4,6,8` die zweite Zeile, und `3,5,7,9` die dritte Zeile initialisiert.

Es sind auch mehr als zwei Dimensionen möglich:

```
int beta[7][3][14][8];
```

In C kann ein n-dimensionales Array stets als ein eindimensionales Array interpretiert werden, dessen Komponenten (n - 1)-dimensionale Arrays sind. Diese Interpretation ist rekursiv, da ein Array auf ein Array mit einer um 1 geringeren Dimension zurückgeführt wird.

Wenn ein Array Elemente hat, die selbst ebenfalls Arrays sind, so gelten die Initialisierungsregeln rekursiv. Das bedeutet, dass die Initialisierungsliste eines mehrdimensionalen Arrays geschachtelte Klammern enthalten.



12.3.4 Erweiterte Initialisierung bei mehrdimensionalen Arrays

Natürlich gelten bei geschachtelten Initialisierungslisten dieselben Regeln wie für eindimensionale Arrays. Es dürfen also nicht mehr Elemente initialisiert werden, als dass das innere Array speichern kann. Es ist aber möglich, Elemente der inneren Initialisierung auszulassen, worauf diese mit 0 gefüllt werden.

Dabei dürfen sowohl Zeilen fehlen als auch Spalten innerhalb einer Zeile unvollständig initialisiert sein. Alle nicht initialisierten Elemente werden mit 0 initialisiert. Natürlich können Initialisierungen nur am Ende einer Zeile weggelassen werden beziehungsweise die letzten Zeilen können ganz fehlen, da ansonsten eine eindeutige Zuordnung der Werte zu den Elementen nicht möglich wäre.

```
int alpha[3][4] = {
  {1},                      // 1 0 0 0
  {1,1}                     // 1 1 0 0
};                          // 0 0 0 0
```

Es ist zudem möglich, geschweifte Klammern von inneren Arrays bei der Initialisierung auch auszulassen. Beginnt die Initialisierung des inneren Arrays nicht mit einer linken geschweiften Klammer, dann werden nur genügend Initialisierungen für die Bestandteile des inneren Arrays aus der Liste entnommen. Sind noch Initialisierungswerte übrig, so werden sie für das nächste Element des äußeren Arrays herangezogen. Die folgende beiden Initialisierung sind somit äquivalent:

```
int alpha[3][4] = {
  { 1, 3, 5, 7 },
  { 2, 4, 6, 8 },
  { 3, 5, 7, 9 },
};

float b[3][4] = {1, 3, 5, 7, 2, 4, 6, 8, 3, 5, 7, 9};
```

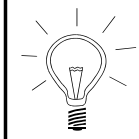
Bei der Initialisierung von b werden hier 4 Elemente für die Initialisierung von `b[0]` benutzt, die nächsten 4 für die Initialisierung von `b[1]` und die letzten 4 für die Initialisierung von `b[2]`. Damit ist das Array mit denselben Werten Initialisiert worden wie `alpha`.

12.3.5 String-Konstanten

Eine **String-Konstante** (ein sogenanntes „**String-Literal**“) besteht aus einer Folge von Zeichen, die in Anführungszeichen eingeschlossen sind. Siehe auch Kapitel → 6.5.4.

```
"Hallo Welt!"
```

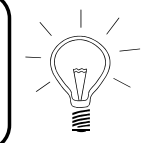
> Eine String-Konstante wird vom Compiler intern als ein Array von Zeichen gespeichert. Dabei wird als letztes Element des Arrays automatisch ein zusätzliches Zeichen, das Zeichen `'\0'` (**Nullzeichen**), angehängt, um das Stringende anzuzeigen.



Die Speicherung eines Strings mit einem zusätzlichen Nullzeichen am Ende wird umgangssprachlich als „**C-String**“ bezeichnet.

Wie jede andere Konstante auch stellt eine String-Konstante einen Ausdruck dar, der nicht verändert, aber gelesen werden kann.

> Der Rückgabewert einer String-Konstanten ist ein Pointer auf das erste Zeichen des Strings. Der Typ des Rückgabewertes ist `const char*`.



12.3.6 `char`-Arrays

`char`-Arrays werden verwendet, um Strings abzuspeichern:

```
char array[20];
```

Im Gegensatz zu String-Konstanten können `char`-Arrays auch verändert werden. In einem solchen Array können beispielsweise Zeichen einzeln mit Werten belegt werden:

```
array[0] = 'a';
```

Initialisiert man ein char-Array manuell sofort bei seiner Definition, so kann – wie bei jedem eindimensionalen Array – eine Initialisierungsliste in geschweiften Klammern verwendet werden. Die einzelnen Werte der Liste – hier die Zeichen – werden durch Kommas getrennt wie im folgenden Beispiel:

```
char array[20] = {'Z', 'e', 'i', 'c', 'h', 'e', 'n',
                  'k', 'e', 't', 't', 'e', '\0'};
```

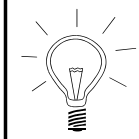
Da das Anschreiben einer Initialisierungsliste als Array von Zeichen mit den vielen Hochkommas sehr mühsam ist, kann man ein char-Array auch mit einer String-Konstanten initialisieren.

Dies würde im Falle des obigen Beispiels dann folgendermaßen aussehen:

```
char array[20] = "Zeichenkette";
```

Beide manuellen Initialisierungen sind vom Speicherinhalt her äquivalent. Die zweite Formulierung stellt eine Abkürzung für die erste, längere Schreibweise dar. Die zweite Form der Initialisierung stellt allerdings einen Sonderfall dar, den es nur für eindimensionale Arrays von Zeichen gibt. Das Array wird mit den Zeichen der String-Konstante initialisiert, wobei ein zusätzliches, abschließendes Nullzeichen '\0' automatisch angehängt wird.

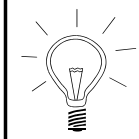
Eine direkte Zuweisung eines Strings an ein eindimensionales Array kann nur bei der Initialisierung erfolgen. Im weiteren Programmablauf sind spezielle Bibliotheksfunktionen für diese Zwecke notwendig.



12.4 Übergabe von Arrays an Funktionen

Bei der Übergabe eines **Arrays als Parameter** an eine Funktion wird als aktueller Parameter der Arrayname angegeben. Dieser wird bei Aufruf der Funktion in einen Pointer auf das erste Element des Arrays ausgewertet.

> Der formale Parameter für die Übergabe eines eindimensionalen Arrays kann ein Array ohne Längenangabe sein, also ein Array mit leeren eckigen Klammern. Da ein Arrayname jedoch bei einem Funktionsaufruf als Pointer ausgewertet wird, kann auch ein Pointer auf den Komponenten-Typ des Arrays als Typ des formalen Parameters verwendet werden.



Ein Array ohne Längenangabe hat jedoch – wie der Name schon sagt – keine explizite Länge.

Um die Länge eines Arrays automatisch zu ermitteln, gibt es die Möglichkeit, eine Endemarkierung als letztes Element des Arrays zu setzen, beispielsweise den Wert `NULL`. Dieser als „Wächter“ oder auf Englisch „sentinel“ bezeichnete Marker kann daraufhin im Code verwendet werden, um die tatsächliche Länge des Arrays bei Bedarf automatisch zu ermitteln. Dies wird beispielsweise bei der Verarbeitung von Strings (siehe Kapitel → 12.9.5) häufig gemacht.

Eine weitaus einfachere Methode, um die Länge eines als Parameter übergebenen Arrays zu ermitteln, ist das Übergeben der Länge als zusätzlichen Parameter.

Folgendes Beispiel zeigt zwei Methoden, ein Array per Parameter zu übergeben:

arrayparameter.c

```
#include <stdio.h>
#define GROESSE 3

void init(int*, int);
void ausgabe(int[], int);

int main(void) {
  int i[GROESSE];
  init(i, GROESSE);
  ausgabe(i, GROESSE);
  return 0;
}
```

```
void init(int* alpha, int dim) {
  for (int i = 0; i < dim; ++i) {
    *alpha = i;
    ++alpha;
  }
}

void ausgabe(int alpha[], int dim) {
  for (int i = 0; i < dim; ++i) {
    printf("i[%d] hat den Wert: %d\n", i, alpha[i]);
  }
}
```

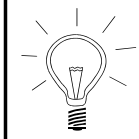
Das Programm erzeugt folgende Ausgabe:

```
i[0] hat den Wert: 0
i[1] hat den Wert: 1
i[2] hat den Wert: 2
```

Da es sich bei dem formalen Parameter `alpha` der Funktion `init()` um einen Pointer handelt, ist somit auch möglich, nur einen Teil des Arrays anzusprechen, indem einfach ein Pointer auf das erste Element des Teil-Arrays und die gewünschte Anzahl der Komponenten übergeben wird.

12.4.1 Übergabe von Strings

Da Strings vom Compiler intern als `char`-Arrays gespeichert werden, ist die Übergabe von **Strings als Parameter** identisch mit der Übergabe von `char`-Arrays. Der formale Parameter einer Funktion, der ein String übergeben bekommt, kann vom Typ `char*` oder `char[]` sein.



Dadurch, dass in einem String ein Nullzeichen das Ende definiert, benötigen Funktionen, die mit Strings arbeiten, keinen zusätzlichen Parameter mit einer Längenangabe. Falls nötig, kann die Länge des Strings über die Funktion `strlen()` (siehe Kapitel → 12.9) berechnet werden.

Es ist jedoch zu beachten, dass String-Konstanten stets mit dem Typ `const char*` übergeben werden müssen.

12.4.2 Ausgabe von Strings und von char-Arrays

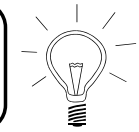
Der Rückgabewert eines Strings ist ein Pointer auf das erste Element des Strings. Damit ist klar, was bei der Übergabe eines Strings an die Funktion `printf()` wie im folgenden Beispiel passiert:

```
printf("Hello, world");
```

Die String-Konstante `"Hello, world"` wird beim Laden des Programmes in den Speicher geladen. An die Funktion `printf()` wird ein Pointer auf diesen String übergeben. So ist `printf()` in der Lage, den Inhalt des Arrays auszudrucken. Die Funktion `printf()` druckt beginnend vom ersten Zeichen alle Zeichen des Arrays aus, bis sie ein Nullzeichen `'\0'` findet.

Bekanntermaßen kann man mit der Funktion `printf()` formatierte Strings ausgeben (daher auch der Name: `printf` = print formatted). Nebst den bekannten Formatelementen `%d` und `%f` für Integer- und Gleitpunkt-Zahlen gibt es jedoch auch das Formatelement `%s`.

String-Variablen werden bei `printf()` mit Hilfe des Formatelements `%s` ausgegeben.



Das Umwandlungszeichen `s` steht für „String“ und die Funktion erwartet als Parameter einen String:

```
char adjectiv[] = "beautiful";
printf("Hello, %s world", adjectiv);
```

Wie man sieht, können so String-Variablen ausgegeben werden.

12.5 Unterschiede zwischen char-Arrays und Pointern auf char

In den bisherigen Kapiteln wurden **char-Arrays** und **String-Konstanten** behandelt. Hier sollen nun die Unterschiede weiter verdeutlicht werden. Wir betrachten die folgenden beiden Variablen:

```
const char* string1   = "Hallo";
      char  string2[] = "Hallo";
```

Die Variable `string1` ist ein Pointer auf eine String-Konstante. Sie speichert selbst also nur eine Adresse, nicht den Inhalt selbst.

Die Variable `string2` ist ein `char`-Array. Es definiert mittels impliziter Längenbestimmung die Anzahl Elemente und speichert genau so viele `char`-Elemente.

Selbst wenn die Typen dieser beiden Variablen unterschiedlich sind, so können sie doch sehr häufig genau gleich als Argumente oder allgemein in Ausdrücken verwendet werden. Dies deswegen, da ein Arrayname dann zu einem Pointer auf das erste Element ausgewertet wird.

Die beiden folgenden Zeilen lassen somit keinen Unterschied erahnen:

```
printf("%s, Welt", string1);
printf("%s, Welt", string2);
```

Trotzdem ist es wichtig, zu begreifen, was hinter diesen beiden Variablen steckt.

Der String "`Hallo`" wird direkt im Programmcode geschrieben, was im Englischen als „string literal“ bezeichnet wird. Dieser String wird bei Programmstart vom Lader in den Speicher geladen und ist dort verfügbar. Allerdings befindet sich dieser String in einem geschützten Bereich des Speichers, auf welchen nur lesend zugegriffen werden darf. Dies ist das sogenannte „Code-Segment“, siehe Kapitel → 15.2.

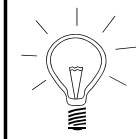
Der der Variablen `string1` zugewiesene Pointer zeigt nach der Initialisierung eben genau auf diesen Speicherbereich, auf welchen der Compiler keine Schreibzugriffe erlaubt. Dementsprechend muss der Typ der Variable `string1` mit `const` deklariert werden.

Unter C ist das Weglassen von const bei der Vereinbarung des Pointers erlaubt. Wenn jedoch auf den Inhalt dieses Pointers ein Schreibzugriff erfolgt, stürzt das Programm auf heutigen Betriebssystemen normalerweise ab. Unter C++ gibt der Compiler einen Fehler aus, wenn versucht wird, das const wegzulassen.

Es ist jedoch erlaubt, die Variable string1 selbst zu verändern, sprich einen anderen Pointer zuzuweisen:

```
string1 = "Guten Morgen";
```

Zeigt ein Pointer auf eine String-Konstante, so ist eine Änderung des Speicherinhalts, auf welchen dieser Pointer zeigt, nicht erlaubt. Die Zuweisung eines neuen Pointers hingegen ist möglich.



Bei der Initialisierung des char-Arrays string2 wird der String "Hallo" explizit in das Array hineinkopiert. Somit existieren nach einer Initialisierung zwei Strings im Speicher: Die String-Konstante, welche vom Lader in den Speicher geladen wurde, sowie der nicht konstante String in dem Array string2.

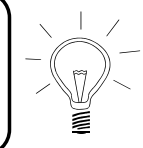
Die Elemente des Arrays string2 sind jedoch nicht mit const deklariert und können somit beliebig angesprochen und verändert werden:

```
string2[1] = 'e';    \\ Hallo -> Hello
```

Jedoch ist es nicht möglich, die Variable selbst auf einen anderen Speicherort zeigen zu lassen. Folgende Anweisung erzeugt somit einen Fehler:

```
string2 = "C-Programming";     // Fehler!
```

Bei einem char-Array string2 kann der in ihm gespeicherte String verändert werden. Die Variable string2 selbst hingegen kann nicht verändert werden.



12.6 Das Schlüsselwort const bei Pointern und Arrays

Wie in Kapitel → 7.5.1 bereits angesprochen, können mithilfe des Schlüsselworts const schreibgeschützte Variablen vereinbart werden.

Die Deklaration mit **const** kann auch auf zusammengesetzte Datentypen wie Arrays angewendet werden:

```
const int feld[] = {1, 2, 3};
```

Hier bedeutet die const-Deklaration, dass alle Feldelemente feld[0], feld[1] und feld[2] nicht überschrieben werden dürfen.

Aufpassen muss man bei der Anwendung des Schlüsselwortes const im Zusammenhang mit Pointern. Angenommen, ein char-Array sei folgendermaßen definiert:

```
char aussage[] = "Pointer sind wichtig.";
```

Dann bedeutet das Pointer-Sternchen * in der folgenden Zeile nicht, dass der Pointer text schreibgeschützt ist, sondern dass dieser Pointer auf einen schreibgeschützten String zeigt.

```
const char* text = "blick";
```

Demnach wird in der folgenden Zeile der Compiler einen Fehler ausgeben:

```
text[1] = 's';
```

Der Pointer jedoch kann auf einen anderen konstanten String zeigen, wie beispielsweise:

```
text = "Jetzt blicke auch ich durch.";
```

Soll tatsächlich der Pointer als schreibgeschützt deklariert werden, so muss const nach dem Pointer-Zeichen stehen wie im folgenden Beispiel:

```
char lili[] = "Ich liebe Lili";
char* const hugo = lili;
```

Man kann sich diese Notation leicht merken, indem man den Typ-Ausdruck von rechts nach links liest mit den Worten „hugo ist ein schreibgeschützter (const) Pointer (*) auf den Typ char“.

In diesem Falle ist dann die folgende Zeile möglich:

```
hugo[13] = 'o';
```

Ob dies Lili gefällt, ist eine andere Frage. Wenigstens kann sie sichergehen, dass folgende Zeile vom Compiler mit einem Fehler geahndet wird:

```
hugo = "Ich liebe Susi";
```

Um hugo stets unzertrennlich mit lili zu verbinden, kann der folgende Typ vereinbart werden:

```
const char* const hugo = lili;
```

Hier sind sowohl der Pointer hugo, als auch der String schreibgeschützt.

Der Schutz eines const-Werts gilt auch für Übergabeparameter, beispielsweise

```
void f(const int* pointer) {
  pointer[3] = 15;             // Fehler !
}
```

Bei Übergabeparametern wird const hauptsächlich zum Schutz der übergebenen Speicherbereiche benutzt (siehe Kapitel → 11.5.2). Eine so definierte Funktion kann auf diese Speicherbereiche nur lesend zugreifen.

Eine Funktion kann auch ein const Ergebnis liefern, wie beispielsweise einen Pointer auf eine String-Konstante. Hierzu muss beim Rückgabetyp der Modifikator const angegeben werden.

```
const char* werLiebtSusi() {
  return "Sag ich nicht.";
}
```

12.7 Beispiel für Strings und Pointerarithmetik 

Normalerweise verwendet man in der Praxis Standardfunktionen zur Stringverarbeitung (siehe Kapitel → 12.9). Hier soll jedoch exemplarisch aufgezeigt werden, wie effizient die Programmiersprache C mit Strings umgehen kann.

Im folgenden Beispiel wird ein String „von Hand“ von einem Array `alpha` in ein Array `beta` kopiert:

manualcopy.c

```
#include <stdio.h>

int main(void) {
  char alpha[30]  = "zu kopierender String";
  char beta[30]   = "";

  int i = 0;
  while (alpha[i] != '\0') {
    beta[i] = alpha[i];
    ++i;
  }
  beta[i] = '\0';

  printf("%s\n", alpha);
  printf("%s\n", beta);

  return 0;
}
```

Die Ausgabe dieses Beispiels lautet:

```
zu kopierender String
zu kopierender String
```

Dieses Beispiel ist sehr klar aufgebaut und sehr gut lesbar. Nun möchten wir dieses Beispiel sukzessive verfeinern. Zuallererst verkürzen wir die Anweisungen zum Kopieren wie folgt:

```
int i = 0;
while ((beta[i] = alpha[i]) != '\0') {
  ++i;
}
```

Hier wurde der Umstand genutzt, dass der Zuweisungs-Operator nicht nur den Wert von `alpha` nach `beta` kopiert, sondern dass der Rückgabewert von dem Zuweisungs-Operator zudem der Wert von `beta[i]` nach der Zuweisung ist. So werden automatisch sämtliche Zeichen kopiert. Erst bei Auftreten des abschließenden `'\0'` wird die Schleife abgebrochen, der Wert `'\0'` wurde jedoch ebenfalls bereits von `alpha` nach `beta` kopiert.

Da nun zudem der Wert von `'\0'` gleich `0` ist und so dem Wahrheitswert „falsch“ entspricht, kann man noch knapper schreiben:

```
int i = 0;
while (beta[i] = alpha[i]) {
  ++i;
}
```

Während bislang das Kopieren mit Hilfe von Array-Komponenten durchgeführt wurde, soll im Folgenden das Kopieren mit Hilfe von Pointern demonstriert werden. In der Pointerschreibweise muss man berücksichtigen, dass `alpha` und `beta` Array-Variablen und deshalb nicht veränderbar sind. Um somit veränderbare Pointer auf die Arrays zu erhalten, müssen die folgenden Pointer-Variablen vereinbart werden:

```
char* ptrAlpha = alpha;
char* ptrBeta  = beta;
```

`ptrAlpha` zeigt auf `alpha[0]` und `ptrBeta` zeigt auf `beta[0]`. Damit kann man nun schreiben:

```
while (*ptrAlpha != '\0') {
  *ptrBeta = *ptrAlpha;
  ptrAlpha++;
  ptrBeta++;
}
*ptrBeta = '\0';
```

Eine knappere Formulierung ist:

```
while (*ptrAlpha != '\0') {
  *ptrBeta++ = *ptrAlpha++;
}
*ptrBeta = '\0';
```

Diese Vereinfachung funktioniert, da der Rückgabewert von `ptrAlpha++` dem Wert vor der Erhöhung um 1 entspricht (Postfix-Operator). Mit dem Operator `*` wird somit der Wert des „aktuellen“ `ptrAlpha` dereferenziert. Entsprechendes gilt für `ptrBeta++`. Nach der Ausdrucksanweisung muss der Nebeneffekt stattgefunden haben, das heißt vor dem nächsten Kopiervorgang zeigen `ptrAlpha` und `ptrBeta` jeweils um ein Zeichen weiter.

Noch kürzer wäre:

```
while (*ptrBeta++ = *ptrAlpha++);
```

Hierbei wird erstens ausgenutzt, dass eine Zuweisung auch einen Nebeneffekt hat, so dass das zugewiesene Zeichen `ptrAlpha` sich nach der Anweisung an der Stelle `ptrBeta` befindet. Zweitens wird die Schleife abgebrochen, wenn ein zu kopierendes Zeichen gleich dem Nullzeichen `'\0'` ist, da der zugewiesene Wert beim Zuweisungs-Operator gleichzeitig als Rückgabewert und somit als Ausdruck für die Bedingung der `while`-Schleife dient.

Das Nullzeichen `'\0'` ist das letzte Zeichen vor dem Abbruch der Iteration, das kopiert wurde. Man spart sich also sogar noch die Zuweisung des Nullzeichens nach der Schleife!

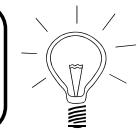
Wie diese Beispiele auch zeigen, benötigt man im Gegensatz zur Array-Schreibweise bei der Formulierung mit Pointern die Laufvariable `i` nicht mehr. Dafür aber die beiden Pointer `ptrAlpha` und `ptrBeta`.

12.8 Funktionen zur Speicherbearbeitung

In diesem Unterkapitel werden Funktionen zur Speicherbearbeitung betrachtet. Um sie zu verwenden, wird die Bibliothek `<string.h>` benötigt.

Speicherbearbeitungsfunktionen arbeiten auf sogenannten „**Puffern**“.

> Als Puffer (auf Englisch „buffer“) wird ganz allgemein ein Speicher bezeichnet, der Daten zwischenspeichert.



Im Rahmen der Speicherbearbeitungsfunktionen entspricht ein Puffer einem Pointer auf einen Ort im Speicher, beispielsweise auf ein Array. Die formalen Parameter der Funktionen erwarten jeweils ein Objekt vom Typ `void*`. Zudem erwarten die Funktionen die Angabe einer entsprechenden Pufferlänge, gemessen in Anzahl Bytes. Die so definierten Puffer-Objekte werden von den Funktionen byteweise behandelt.

In der Standardbibliothek von C existieren die folgenden Funktionen:

`memcpy()`	Kopieren von Puffern
`memmove()`	Überlappendes Kopieren von Puffern
`memcmp()`	Vergleichen von Puffern
`memchr()`	Suchen nach einem Bytewert
`memset()`	Initialisieren eines Puffers

Zu jeder Funktion wird jeweils zuallererst der Prototyp hingeschrieben und danach die Funktionalität vorgestellt.

12.8.1 Die Funktion `memcpy()`

```
void* memcpy(void* dest, const void* src, size_t n);
```

Die Funktion `memcpy()` kopiert `n` Bytes aus dem Puffer, auf den der Pointer `src` zeigt, in den Puffer, auf den der Pointer `dest` zeigt.

Ist der zu kopierende Puffer größer als der Zielpuffer, dann werden nachfolgende Speicherobjekte überschrieben!

Der Rückgabewert der Funktion `memcpy()` ist der Pointer `dest`.

Ab dem Standard C99 werden die beiden Puffer-Parameter mit dem `restrict`-Schlüsselwort deklariert, siehe Kapitel → 8.4. Überlappen sich die Puffer, auf die die Pointer `src` und `dest` zeigen, so ist das Ergebnis undefiniert.

Beispiel:

memcpy.c

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

int main(void) {
  int array1 [10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int array2 [3]  = {333, 444, 555};
  memcpy(array1 + 2, array2, sizeof(array2));

  for (size_t i = 0; i < 10; ++i) {
    printf("%d, ", array1[i]);
  }

  return 0;
}
```

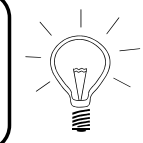
Die Ausgabe ist:

```
Ergebnis: 1, 2, 333, 444, 555, 6, 7, 8, 9, 10
```

12.8.2 Die Funktion `memmove()`

```
void* memmove(void* dest, const void* src, size_t n);
```

> Die Funktion `memmove()` kopiert `n` Bytes von einem Puffer in einen anderen – und zwar auch bei überlappenden Speicherbereichen korrekt.



Im Gegensatz zur Funktion `memcpy()` ist bei der Funktion `memmove()` sichergestellt, dass bei überlappenden Speicherbereichen das korrekte Ergebnis erzielt wird. Der Puffer, auf den der Pointer `src` zeigt, kann dabei überschrieben werden.

> Ist der zu kopierende Puffer größer als der Zielpuffer, dann werden nachfolgende Speicherobjekte überschrieben!

Der Rückgabewert der Funktion `memmove()` ist der Pointer `dest`.

Beispiel:

`memmove.c`

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

int main(void) {
  char string[] = "12345678";
  memmove(string + 2, string, strlen(string) - 2);
  printf("Ergebnis: %s\n", string);
  return 0;
}
```

Ausgabe:

```
Ergebnis: 12123456
```

Würde dieses Beispiel mit `memcpy()` anstelle `memmove()` erfolgen, so ist das Ergebnis undefiniert. Es könnte sein, dass `memcpy()` eine einfache Schleife implementiert, welche die Inhalte Byte für Byte kopieren. Dann könnte das Ergebnis auch lauten:

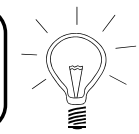
```
Ergebnis: 12121212
```

Dies deswegen, da die Kopierfunktion bei dem Byte `string[4]` den Inhalt von `string[2]` kopiert, welcher bereits vorgängig von `string[0]` überschrieben wurde. Für diese Problematik gibt es eine Übungsaufgabe am Ende dieses Kapitels.

12.8.3 Die Funktion `memcmp()`

```
int memcmp(const void* s1, const void* s2, size_t n);
```

Die Funktion `memcmp()` vergleicht `n` Bytes aus zwei verschiedenen Puffern.



Die Funktion `memcmp()` führt einen byteweisen Vergleich der ersten `n` Bytes der an die Pointer `s1` und `s2` übergebenen Puffer durch. Die Puffer werden solange verglichen, bis entweder ein Byte unterschiedlich oder die Anzahl `n` Bytes erreicht ist.

Die Funktion `memcmp()` gibt folgende Rückgabewerte zurück:

- `< 0` wenn das erste Byte, das in beiden Puffern verschieden ist, im Puffer, auf den der Pointer `s1` zeigt, einen kleineren Wert hat.
- `== 0` wenn `n` Bytes der beiden Puffer gleich sind.
- `> 0` wenn das erste in beiden Puffern verschiedene Byte im Puffer, auf den der Pointer `s1` zeigt, einen größeren Wert hat.

Beispiel:

memcmp.c

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

int main(void) {
  char string1[] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06};
  char string2[] = {0x01, 0x02, 0x03, 0x14, 0x05, 0x06};
  int cmp = memcmp(string1, string2, sizeof(string1));
  printf("Vergleich String1 mit String2 ergibt: %d \n", cmp);
  return 0;
}
```

Die Ausgabe ist beispielsweise:

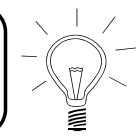
```
Vergleich String1 mit String2 ergibt: -1
```

Die Bildung des Rückgabewertes ist compilerabhängig. Manche Compiler geben den rechnerischen Unterschied des ersten unterschiedlichen Bytes aus, andere geben lediglich eine positive oder negative Zahl aus, was nach dem ISO-Standard genügt.

12.8.4 Die Funktion `memchr()`

```
void* memchr(const void* s, int c, size_t n);
```

Die Funktion `memchr()` durchsucht einen Puffer nach einem bestimmten Byte.



Die Funktion `memchr()` durchsucht die ersten `n` Bytes des Puffers, auf den der Pointer `s` zeigt, nach dem Wert von `c`. Dabei werden der Wert `c` wie auch alle `n` Bytes des Puffers als `unsigned char` interpretiert.

Wird der Wert von `c` gefunden, so gibt die Funktion `memchr()` einen Pointer auf das erste Vorkommen im Puffer zurück, auf den der Pointer `s` zeigt. Ist der Wert von `c` in den ersten `n` Bytes nicht enthalten, so wird ein `NULL`-Pointer zurückgegeben.

Beispiel:

memchr.c

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

int main(void) {
  char str1 [] = "Zeile1: Text";
  char* str2 = memchr(str1, ':', strlen(str1));
  if (str2 != NULL) {
    printf("%s\n", str2);
  }
  return 0;
}
```

Die Ausgabe ist:

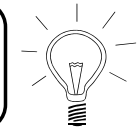
```
: Text
```

Die Funktion `memchr()` erwartet den Puffer mit dem Typ `const void*`, gibt jedoch einen Wert vom Typ `void*` zurück. Ein Schreibzugriff auf den Inhalt des Rückgabewertes kann unerwünschte Nebeneffekte hervorrufen oder gar zum Absturz des Programmes führen.

12.8.5 Die Funktion `memset()`

```
void* memset(void* s, int c, size_t n);
```

Die Funktion `memset()` setzt die ersten `n` Bytes eines Puffers auf den Wert eines vorgegebenen Bytes `c`.



Dabei werden der Wert `c` wie auch alle `n` Bytes des Puffers als `unsigned char` interpretiert.

Der Rückgabewert der Funktion `memset()` ist der Pointer `s`.

Beispiel:

memset.c

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

int main(void) {
  char string[20] = "Hallo";
  printf("Ergebnis: %s\n", memset(string, '*', 5));
  return 0;
}
```

Die Ausgabe ist:

```
Ergebnis: *****
```

12.9 Standardfunktionen zur Stringverarbeitung

Der C-Standard definiert mehrere Funktionen, welche die Manipulation von Strings ermöglichen. Um sie zu verwenden, wird die Bibliothek `<string.h>` benötigt.

Stringverarbeitungsfunktionen funktionieren fast gleich wie die Funktionen zur Speicherbearbeitung. Im Gegensatz dazu berücksichtigen String-Funktionen hingegen zusätzlich noch das Nullzeichen `'\0'` am Ende eines Strings.

In diesem Kapitel werden nur die einfacheren Funktionen behandelt. In den Standardbibliotheken gibt es noch viele weitere Funktionen, welche beispielsweise mit wide characters umgehen oder einzelne Zeichen oder gar ganze Strings innerhalb eines Strings suchen können. Im C11-Standard wurden zudem für bestimmte Funktionen zusätzlich sogenannte „sichere“ Funktionen definiert, welche eine Prüfung der übergebenen Pufferlänge ermöglichen. Die Gesamtheit aller verfügbaren Funktionen kann an dieser Stelle nicht angesprochen werden.

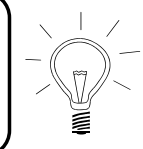
Folgende Stringverarbeitungsfunktionen werden vorgestellt:

`strcpy()`	Kopieren von Strings
`strncpy()`	Kopieren von Strings mit Längenangabe
`strcat()`	Anhängen eines Strings an einen anderen
`strcmp()`	Vergleichen von Strings
`strlen()`	Ermitteln der Stringlänge

12.9.1 Die Funktion `strcpy()`

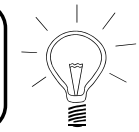
```
char* strcpy(char* dest, const char* src);
```

> Die Funktion `strcpy()` kopiert den Inhalt des Strings, auf den der Pointer `src` zeigt, an die Adresse, auf die der Pointer `dest` zeigt.



Die Zieladresse muss hierbei an einen Ort im Speicher zeigen, an welchem genügend Platz reserviert ist, um den gesamten String inklusive das abschließende Stringende-Zeichen `'\0'` zu fassen.

> Kopiert wird der gesamte Inhalt einschließlich des Stringende-Zeichens `'\0'`.



Die Funktion `strcpy()` überprüft dabei nicht, ob der Puffer, dessen Adresse übergeben wurde, genügend Platz zur Verfügung stellt. Hierfür muss man selbst Sorge tragen.

> Ist der zu kopierende Puffer größer als der Zielpuffer, dann werden nachfolgende Speicherobjekte überschrieben!

Seit C99 werden die beiden Parameter mit dem `restrict`-Schlüsselwort deklariert, siehe Kapitel → 8.4. Dies bedeutet, dass wenn zwischen zwei sich überlappenden Objekten kopiert wird, das Verhalten undefiniert ist. Im obigen Prototyp wurde auf die Angabe des Schlüsselworts verzichtet.

Die Funktion `strcpy()` gibt als Rückgabewert den Pointer `dest` zurück.

Folgendes ist ein einfaches Programmbeispiel:

strcpy.c

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

int main(void) {
  char string1[25];
  char string2[] = "Zu kopierender String";
  strcpy(string1, string2);
  printf("Der kopierte String ist: %s\n", string1);
  return 0;
}
```

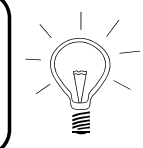
Das Beispiel erzeugt die folgende Ausgabe:

```
Der kopierte String ist: Zu kopierender String
```

12.9.2 Die Funktion `strncpy()`

```
char* strncpy(char* dest, const char* src, size_t n);
```

Die Funktion `strncpy()` kopiert genauso wie `strcpy()` einen String an eine andere Adresse, wird jedoch garantiert nicht mehr als eine vorgegebene Anzahl Zeichen kopieren.



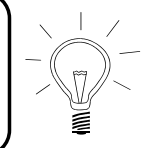
Der Parameter `n` der Funktion `strncpy()` kann somit genutzt werden, um sicherzustellen, dass der Puffer `dest`, dessen Adresse übergeben wurde, nicht durch das Kopieren überquillt.

Dabei wird das Stringende-Zeichen `'\0'` bei den `n` Zeichen mitgezählt.

Hat der zu kopierende String (inklusive dem Stringende-Zeichen `'\0'`) mehr als `n` Zeichen, so wird die Kopie beim `n`-ten Zeichen abgebrochen. Der String an der Adresse `dest` ist in diesem Falle nicht mit einen Stringende-Zeichen `'\0'` abgeschlossen.

Wird als Parameter n eine Zahl angegeben, welche größer ist als der reservierte Speicherplatz an der Stelle des Puffers dest, werden die darauffolgenden Speicherbereiche überschrieben!

Hat der zu kopierende String weniger Zeichen als n Zeichen, so werden die überschüssigen Zeichen in dest mit dem Stringende-Zeichen '\0' aufgefüllt.



Wie schon bei strcpy() werden seit C99 die beiden Pointer-Parameter mit dem restrict-Schlüsselwort deklariert, siehe Kapitel → 8.4. Dies bedeutet, dass wenn zwischen zwei sich überlappenden Objekten kopiert wird, das Verhalten undefiniert ist. Im obigen Prototyp wurde auf die Angabe des Schlüsselworts verzichtet.

Die Funktion strncpy() gibt als Rückgabewert den Pointer dest zurück.

Ein einfaches Beispiel:

strncpy.c

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

int main(void) {
  char string1[] = "XXXXXXXXXXXXXXXXXXXXXXX";
  char string2[] = "Zu kopierender String";
  strncpy(string1, string2, 10);
  printf("Der kopierte String ist: %s\n", string1);
  return 0;
}
```

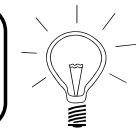
Folgendes ist die Ausgabe:

```
Der kopierte String ist: Zu kopiereXXXXXXXXXXXXX
```

12.9.3 Die Funktion `strcat()`

```
char* strcat(char* dest, const char* src);
```

> Die Funktion `strcat()` hängt einen String an einen anderen an.



Die Funktion `strcat()` hängt an den String, auf den der Pointer `dest` zeigt, den String an, auf den der Pointer `src` zeigt. Dabei wird das Stringende-Zeichen `'\0'` des Strings, auf den der Pointer `dest` zeigt, vom ersten Zeichen des Strings, auf den der Pointer `src` zeigt, überschrieben. Daraufhin wird der restliche String kopiert, auf den der Pointer `src` zeigt, einschließlich des Zeichens `'\0'`.

Der Name der Funktion kommt von dem englischen „concatenate“ = zusammenfügen.

> Die Funktion `strcat()` prüft nicht, ob genügend Speicher im String, auf den der Pointer `dest` zeigt, vorhanden ist. Die Kontrolle des zur Verfügung stehenden Speichers muss man selbst verantworten. Reicht der Puffer nicht aus, werden nachfolgende Speicherobjekte überschrieben.

Auch hier gilt, dass seit C99 die beiden Pointer-Parameter mit dem `restrict`-Schlüsselwort deklariert werden, siehe Kapitel → 8.4. Dies bedeutet, dass wenn die beiden Speicherbereiche sich überlappenden, das Verhalten undefiniert ist. Im obigen Prototyp wurde auf die Angabe des Schlüsselworts verzichtet.

Der Rückgabewert der Funktion `strcat()` ist ein Pointer auf den zusammengefügten String, also der Pointer `dest`.

Beispiel:

strcat.c

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

int main(void) {
   char string[50] = "concatenate";
   strcat(string, " = zusammenfuegen");
   printf("%s\n", string);
   return 0;
}
```

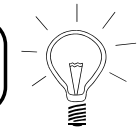
Die Ausgabe ist:

```
concatenate = zusammenfuegen
```

12.9.4 Die Funktion `strcmp()`

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

Die Funktion `strcmp()` vergleicht zwei Strings zeichenweise.



Die Funktion `strcmp()` führt einen zeichenweisen Vergleich der beiden Strings durch, auf die die Pointer `s1` und `s2` zeigen. Sie werden solange verglichen, bis ein Zeichen unterschiedlich oder bis ein Stringende-Zeichen `'\0'` erreicht ist.

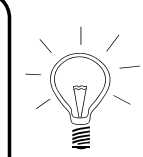
Die Funktion `strcmp()` gibt einen `int`-Wert zurück mit folgender Bedeutung:

- `< 0` wenn der String, auf den der Pointer `s1` zeigt, lexikografisch kleiner ist als der String, auf den der Pointer `s2` zeigt.
- `== 0` wenn der String, auf den der Pointer `s1` zeigt, lexikografisch gleich dem String ist, auf den der Pointer `s2` zeigt.
- `> 0` wenn der String, auf den der Pointer `s1` zeigt, lexikografisch größer ist als der String, auf den der Pointer `s2` zeigt.

Der Begriff „lexikographisch“ bedeutet hierbei, dass die Zeichen gemäß der Stellung im Alphabet bewertet werden. Anders gesagt, steht der Buchstabe A vor dem Buchstaben B, er ist also lexikographisch kleiner als B.

Die Werte `< 0` beziehungsweise `> 0` entstehen durch den Vergleich zweier unterschiedlicher `unsigned char`-Zeichen.

> Der Vergleich von zwei Strings mit der Methode `strcmp()` erfolgt durch einen Vergleich der einzelnen Zeichen an den äquivalenten Positionen in den beiden Strings, und zwar von vorne nach hinten.



Beim Vergleich werden die entsprechenden Zeichen voneinander subtrahiert. Sobald das Ergebnis der Subtraktion zweier Zeichen eine Zahl ungleich 0 ist, wird der Vergleich abgebrochen. Sind die Strings ungleich lang, so wird bis zum Stringbegrenzungszeichen `'\0'` des kürzeren Strings verglichen.

Beispiel:

strcmp.c

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

int main(void) {
  printf("%d\n", strcmp("abcde", "abCde"));
  printf("%d\n", strcmp("abcde", "abcde"));
  printf("%d\n", strcmp("abcd",  "abcde"));
  return 0;
}
```

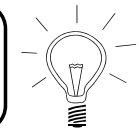
Die Ausgabe lautet:

```
1
0
-1
```

12.9.5 Die Funktion `strlen()`

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

> Die Funktion `strlen()` bestimmt die Anzahl Zeichen eines Strings.



Die Funktion `strlen()` liefert als Rückgabewert die Anzahl der Zeichen des Strings, auf den der Pointer s zeigt. Das Stringende-Zeichen `'\0'` wird dabei nicht mitgezählt.

Beispiel:

strlen.c

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

int main(void) {
  char string[] = "Programm";
  printf("Das Array hat %d Elemente.\n", (int)sizeof(string));
  printf("Der String hat %d Zeichen.\n", (int)strlen(string));
  return 0;
}
```

Die Ausgabe ist:

```
Das Array hat 9 Elemente.
Der String hat 8 Zeichen.
```

Dieses Programm zeigt den Unterschied zwischen `sizeof` und `strlen()` auf. Mit `sizeof` wird die Größe des Arrays ermittelt und mit `strlen()`, wie weit es aktuell gefüllt ist.

> Die Funktion `strlen()` liefert die Länge eines Strings zurück, ohne das Zeichen `'\0'` mitzuzählen. Will man beispielsweise prüfen, ob ein `char`-Array groß genug zur Aufnahme eines String ist, dann muss man noch `+1` zur Länge des Strings dazu addieren.

12.10 Weitere Funktionen in <string.h>

Die verschiedenen Implementationen der Standardbibliothek (beispielsweise POSIX, GNU, BSD, ...) boten über die Jahre noch weitere Funktionen an. Diese als Extensions (Erweiterungen) bezeichneten Funktionen werden in diesem Buch nicht weiter beschrieben.

Nennenswert sind jedoch die Funktionen, welche in dem kommenden Standard C23 eingeführt werden. Sie werden hier kurz aufgelistet:

`void* memset_explicit(void* dest, int ch, size_t count)`
Stellt sicher, dass sämtliche Speicherinhalte mit einem gegebenen Zeichen überschrieben werden.

`void* memccpy(void* restrict dest, const void* restrict src, int c, size_t count )`
Kopieren von nicht-überlappendem Speicher bis zum Auftreten eines gegebenen Zeichens.

`char* strdup(const char* str1)`
Duplizieren eines Strings mit Speicherallokation.

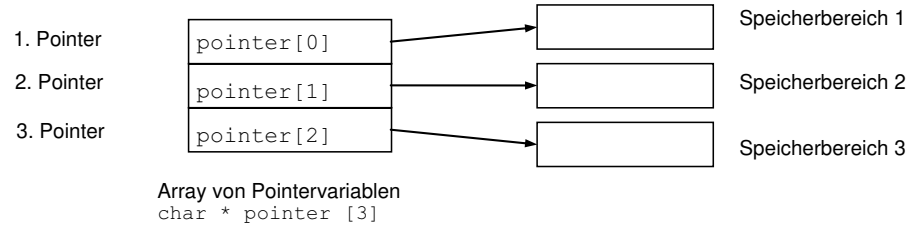
`char* strndup(const char *src, size_t size)`
Duplizieren eines Strings mit einer gegebenen Länge mit Speicherallokation.

12.11 Beispiele für Arrays und Pointer

Um die Inhalte dieses Kapitels zu vertiefen, werden hier weitere Beispiele für Arrays und Pointer aufgeführt.

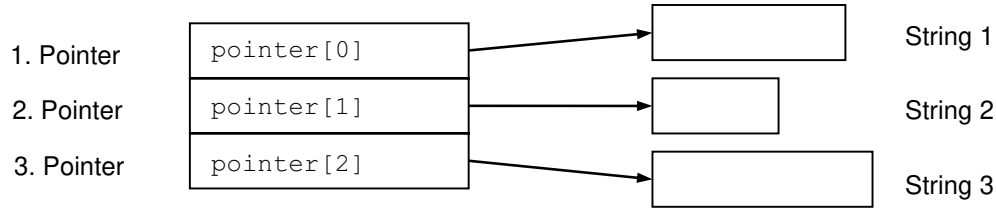
12.11.1 Beispiel für eindimensionale Arrays von Pointern

Ein Pointer ist eine Variable, in der die Adresse eines anderen Speicherobjektes (Variable, Funktion) gespeichert ist. Entsprechend einem eindimensionalen Array von gewöhnlichen Variablen kann natürlich auch ein eindimensionales Array von Pointer-Variablen gebildet werden, wie im folgenden Beispiel eines eindimensionalen Arrays aus 3 Pointern auf `char` zu sehen ist:



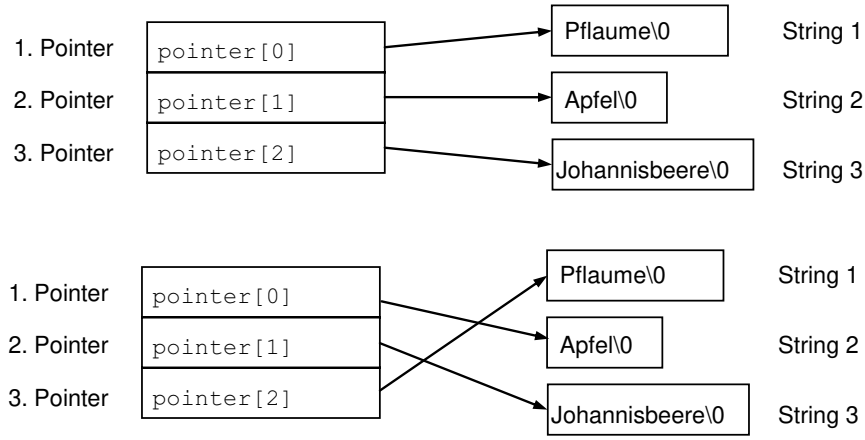
Arbeitet man mit einem String fester Länge, so legt man ein Zeichenarray einer festen Größe an, wie beispielsweise `char a[20]`. Ist die Länge eines Strings von vornherein jedoch nicht fest definiert, so verwendet man meist einen Pointer wie beispielsweise `char* pointer` und lässt den Pointer auf den String zeigen. Arbeitet man mit mehreren Strings, deren Länge nicht von vornherein bekannt ist, so verwendet man ein eindimensionales Array von Pointern auf `char`.

Im folgenden Beispiel stellt `char* pointer[3]` ein eindimensionales Array von drei Pointern auf `char` dar, die auf drei Strings zeigen:



Will man beispielsweise diese Strings sortieren, so muss dies nicht mit Hilfe von aufwendigen Kopieraktionen für die Strings durchgeführt werden. Es werden lediglich die Pointer so verändert, dass die geforderte Sortierung erreicht wird.

Die beiden folgenden Bilder zeigen die Pointer vor und nach dem Sortieren der Strings:



Ein weiteres Beispiel ist die folgende Funktion, welche zeilenweise einen Text auf dem Bildschirm ausgibt:

```
void textausgabe(char* textPointer[], int anzZeilen) {
  for (size_t i = 0; i < anzZeilen; ++i)
    printf("%s\n", textPointer[i]);
}
```

In Kapitel → 12.3.2 wurde erläutert, dass der formale Parameter für die Übergabe eines Arrays in der Notation eines Arrays ohne Längenangaben geschrieben werden kann. Damit ist die Notation `char* textPointer[]` verständlich. Die Angabe `[]` ist jedoch äquivalent zu einem Pointer. Somit könnte man alternativ als aktuellen Parameter `char** textPointer` schreiben, also ein Pointer auf einen Pointer.

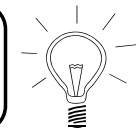
12.11.2 Beispiel für Pointer auf Pointer

Die Variable `pointer` aus Kapitel → 12.11.1 ist ein eindimensionales Array aus drei Elementen. Jedes Element ist ein Pointer auf einen `char`-Wert. Wird einer dieser Pointer dereferenziert, beispielsweise durch `*pointer[i]`, so erhält man das erste Zeichen dieses Strings.

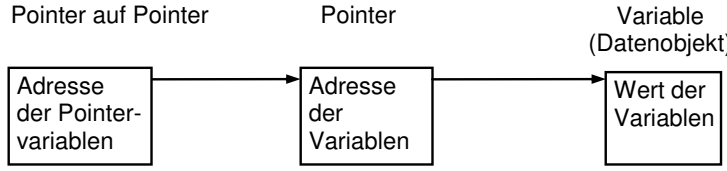
Im Folgenden soll nun das Beispiel etwas anders formuliert werden:

```
void textausgabe(char** textPointer, int anzZeilen) {
  while (anzZeilen-- > 0)
    printf("%s\n", *textPointer++);
}
```

> Die beiden Schreibweisen `char** textPointer` und `char* textPointer[]` sind bei formalen Parametern gleichwertig.



Das folgende Bild soll einen Pointer auf einen Pointer veranschaulichen:



Folgender Aufruf der oben gezeigten Funktion `textausgabe()` demonstriert die Funktionsweise von Pointern auf Pointer:

```
int main(void) {
  char* zeilen[5] = {
    "Mein",
    "Computer",
    "kennt",
    "Else",
    "nicht"
  };
  textausgabe(&zeilen[1], 3);
  return 0;
}
```

Der formale Parameter `char** textPointer` bekommt beim Aufruf der Funktion `textausgabe()` als aktuellen Parameter eine Adresse des ersten Elements eines Arrays übergeben. In diesem Beispiel ist dieses Array das Element `zeilen[1]`, wo ein Pointer auf den String `"Computer"` gespeichert ist. Der dereferenzierte Pointer `*textPointer` wird der Funktion `printf()` übergeben, welche einen Pointer auf `char` erwartet. Gleichzeitig wird mit dem Inkrement-Operator die Variable `textPointer` um eine Einheit erhöht. Somit zeigt `textPointer` nach dem ersten Durchgang der Schleife auf das Element `zeilen[2]`, wo ein Pointer auf den String `"kennt"` gespeichert ist. Die Schleife wird solange fortgesetzt, bis `3` Strings ausgegeben sind.

Der vollständige Programmcode befindet sich in der Datei `elsescomputer.c`. Die Ausgabe des Programmes lautet:

```
Computer
kennt
Else
```

12.11.3 Beispiel für die Initialisierung von Arrays von Pointern

Das folgende Beispiel zeigt eine Funktion, die bei der Übergabe eines Fehlercodes einen Pointer auf den entsprechenden Fehlertext zurückliefert:

fehler.c

```
#include <stdio.h>

const char* fehlertext(int n) {
  static const char* errDesc[] = {
    "kein Fehler",
    "Fehlertext Eins",
    "Fehlertext Zwei",
    "Fehlertext Drei"};

  return (n < 0 || n > 3) ? "Fehlercode existiert nicht" :
         errDesc[n];
}

int main(void) {
  int fehlernummer = 2;
  printf("Text zu Fehler %d: %s\n",
    fehlernummer, fehlertext(fehlernummer));
  return 0;
}
```

Ein Beispiel für einen Programmlauf ist:

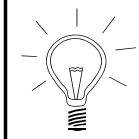
```
Text zu Fehler 2: Fehlertext Zwei
```

Die Funktion `main()` dient hier nur zu Testzwecken. In einem richtigen Programm wird die Funktion `fehlertext()` zur Laufzeit im Fehlerfall von anderen Funktionen aufgerufen, welche die Fehlernummer übergeben und den Fehlertext zurückerhalten, um ihn auszugeben.

Das Array von Pointern auf `const char* errDesc[]` muss als `static` angelegt werden, da hier eine lokale Pointer-Variable aus der Funktion zurückgegeben wird (siehe auch Kapitel → 15.4). Ohne `static` wäre der Rückgabewert der Funktion nach dem Funktionsende undefiniert, da nach Ablauf einer Funktion ihre lokalen Variablen ungültig werden. Mit dem Schlüsselwort `static` bleiben die Fehlertexte als interne Variablen über die gesamte Laufzeit des Programms permanent im Speicher.

12.11.4 Vergleich zweidimensionales gegen eindimensionales Array von Pointern

> Der Unterschied zwischen einem eindimensionalen Array von Pointern und einem zweidimensionalen Array ist, dass bei mehrdimensionalen Arrays die Anzahl der Elemente fest vorgegeben ist, bei eindimensionalen Arrays von Pointern hingegen nur die Anzahl an Pointern.



So definiert im folgenden Beispiel die erste Zeile ein Array mit insgesamt 50 `int`-Elementen und die zweite Zeile ein eindimensionales Array von 5 Pointern auf `int`:

```
int  array2D      [5][10];
int* pointerArray [5];
```

Der Vorteil des Arrays aus Pointern `pointerArray` besteht darin, das die „2-te Dimension“ der einzelnen Elemente des Arrays unterschiedlich groß sein kann. Das heißt im Gegensatz zum `array2D` muss nicht jedes Element 10 `int`-Werte haben.

Die häufigste Anwendung besteht deshalb darin, ein Array unterschiedlich langer Strings zu bilden wie im Falle der Funktion `fehlertext()` in Kapitel → 12.11.3:

```
char* errDesc1[] = {
  "Fehlercode existiert nicht",      // 27 Bytes
  "Fehlertext 1",                    // 13 Bytes
  "Fehlertext 2" };                  // 13 Bytes
```

Mit dieser Definition werden insgesamt 53 Bytes für die Zeichen des Strings benötigt. Zusätzlich benötigen die Pointer noch einige Bytes.

Zum Vergleich:

```
char errDesc2[][27] = {
  "Fehlercode existiert nicht",     // 27 Bytes
  "Fehlertext 1",                   // 27 Bytes
  "Fehlertext 2" };                 // 27 Bytes
```

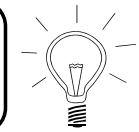
Hier muss mindestens die Anzahl an Elementen reserviert werden, die der längste String benötigt. Dies sind 27 Bytes für `"Fehlercode existiert nicht"`. Die restlichen Strings benötigen zwar nur je 13 Zeichen, für sie sind aber ebenfalls 27 Zeichen reserviert. Somit benötigt `errDesc2` insgesamt 81 Bytes

Die Einsparung an Speicherplatz zeigt sich eindrucksvoll, wenn beispielsweise eine Maximallänge von 255 Zeichen festgelegt wird, die gespeicherten Strings jedoch durchschnittlich nur rund 10 Zeichen benötigen. Werden in einer solchen Struktur 1 Million Strings gespeichert, so benötigt das zweidimensionale Array 255 Millionen Bytes, das Pointer-Array jedoch gerade mal deren 10 Millionen.

12.12 Pointer auf Funktionen

Pointer auf Funktionen sind ein spezieller Pointer-Typ, welcher in einer Variablen die Adresse einer aufzurufenden Funktion speichern kann.

> Über einen Pointer können Funktionen auch als Parameter an andere Funktionen übergeben werden.



Damit kann von Aufruf zu Aufruf eine unterschiedliche Funktion als Argument übergeben werden. Ein bekanntes Beispiel für die Anwendung von Pointern auf Funktionen ist die Übergabe unterschiedlicher Funktionen an eine Sortierfunktion, um beispielsweise aufsteigend oder absteigend zu sortieren.

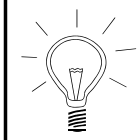
Ein weiteres Einsatzgebiet von Pointern auf Funktionen ist die Programmierung von Interrupts. Eine Interrupt-Behandlung wird ermöglicht, indem man in die Interrupt-Tabelle des Betriebssystems den Pointer auf die Interrupt-Service-Routine (ISR) schreibt. Bei einem Interrupt wird dann die entsprechende Interrupt-Service-Routine aufgerufen.

Das folgende Bild zeigt eine Interrupt-Tabelle:

Pointer auf ISR n
. . .
. . .
Pointer auf ISR 2
Pointer auf ISR 1

12.12.1 Vereinbarung eines Pointers auf eine Funktion

> Ein Funktionsname bezeichnet in C den Adressbereich, in welchem eine Funktion gespeichert ist. Durch Vereinbarung eines Pointers auf eine Funktion wird die Adresse der ersten Anweisung (genauer gesagt, des ersten Maschinenbefehls) einer Funktion gespeichert.



Die Vereinbarung eines Funktionspointers sieht auf den ersten Blick etwas kompliziert aus, wie im folgenden Beispiel:

```
int (*ptr)(char);
```

`ptr` ist hierbei ein Pointer auf eine Funktion mit einem Rückgabewert vom Typ `int` und einem Übergabeparameter vom Typ `char`. Das erste Klammernpaar ist unbedingt nötig, da es sich sonst an dieser Stelle um einen gewöhnlichen Funktions-Prototypen handeln würde. Infolge der Klammern muss man lesen „`ptr` ist ein Pointer auf“. Dann kommen entsprechend der Operatorpriorität die runden Klammern. Also ist „`ptr` ein Pointer auf eine Funktion mit einem Übergabeparameter vom Typ `char`“. Als letztes wird das `int` gelesen, das heißt die Funktion hat den Rückgabetyp `int`.

Wie funktioniert aber nun das Arbeiten mit einem Pointer auf eine Funktion? Dem noch nicht gesetzten Pointer auf eine Funktion muss hierzu eine Adresse einer bekannten Funktion desselben Typs zugewiesen werden, beispielsweise so:

```
ptr = &funktionsname;
```

Der Standard lässt bei der Ermittlung der Adresse einer Funktion auch zu, direkt den Funktionsnamen hinzuschreiben, ohne den Adress-Operator zu verwenden, da ein Funktionsname ähnlich wie eine Array-Variable bei Auftreten im Code automatisch in seine Adresse ausgewertet wird.

```
ptr = funktionsname;
```

12.12.2 Aufruf einer Funktion

Da nun `ptr` (Pointer auf eine Funktion) die Adresse der Funktion `funktionsname()` enthält, kann der Aufruf der Funktion auch durch die Dereferenzierung des Pointers erfolgen. Man beachte im folgenden Beispiel die Klammerung um den dereferenzierten Pointer:

```
int a;
a = funktionsname('A');

ptr = &funktionsname;
a = (*ptr)('A');
```

Im diesem Beispiel sind beide Zuweisungen zur Variablen `a` somit equivalent.

C erlaubt es zudem auch, die Klammerung um die Dereferenzierung wegzulassen, sodass der Funktionspointer wie ein Funktionsname benutzt werden kann:

```
a = ptr('A');
```

12.12.3 Beispiel für Pointer auf eine Funktion

Pointer auf Funktionen werden normalerweise benutzt, um sie als Parameter an Funktionen zu übergeben, die dann über den Pointer auf eine gewünschte Funktionalität zugreifen, welche innerhalb der aufgerufenen Funktion sonst nicht verfügbar wäre. Manche Problemstellungen lassen sich durch den Einsatz von Pointern auf Funktionen elegant lösen.

Im folgenden Programm ist die Funktion `evalTime()` durch die Übergabe eines Pointers auf eine Funktion in der Lage, die Durchlaufzeit jeder übergebenen Funktion passenden Typs elegant zu berechnen, hier in diesem Beispiel die Zeit bis zum Drücken der <RETURN>-Taste:

functionPointer.c

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

void f1(void) {
  printf("Druecken Sie bitte Enter:");
  getchar();
}
```

```
double evalTime(void (*ptr)(void)) {
  time_t begin, end;
  begin = time(NULL);
  ptr();
  end = time(NULL);
  return difftime(end, begin);
}

int main(void) {
  printf("Zeit bis Enter gedrueckt wurde: %1.0f sec\n",
    evalTime(f1));
  return 0;
}
```

Hier ein Beispiel für die Ausgabe des Programms:

```
Druecken Sie bitte Enter:
Zeit bis Enter gedrueckt wurde: 2 sec
```

Die Funktion `time()` liefert als Rückgabewert die aktuelle Kalenderzeit in einer Darstellung, die vom Compiler abhängig ist. Die Funktion `difftime()` berechnet die Zeit zwischen zwei Zeitangaben in Sekunden. Beide Funktionen sind in der Bibliothek <`time.h`> enthalten, vergleiche Kapitel → 13.1.10.

Es ist nun vorstellbar, dass an diese Funktion jeder beliebige Funktionspointer mit der richtigen Signatur übergeben werden kann. So ist es ganz einfach möglich, die Laufzeit beispielsweise einer komplizierten Berechnung zu messen.

Ein weiteres Beispiel für Pointer auf Funktionen im Zusammenspiel mit der Bibliotheksfunktion `qsort()` ist in Kapitel → 20.1.6 zu finden.

12.13 Pointer auf `void` für Strukturen 

Für dieses Unterkapitel lohnt es sich, erst die Strukturen in Kapitel → 13 nachzulesen.

In diesem Kapitel sowie im Kapitel → 8.2 wurde bereits mehrfach auf `void`-Pointer hingewiesen. Es handelt sich um einen Pointer auf einen unbestimmten Typ. So wurde der Typ `void*` bereits als Parameter oder Rückgabewert einer Funktion verwendet, um auf Speicherbereiche wie Arrays oder Strings zuzugreifen.

Hier in diesem Unterkapitel soll nun ein fortgeschrittenes Beispiel gezeigt werden, bei welchem ein `void`-Pointer auf eine sogenannte „Struktur“ zeigt. Grob umrissen speichert eine Struktur mehrere Attribute eines Objektes in einem zusammengefassten Datentyp. Mittels des `void`-Pointer können solche Strukturen adressiert werden, ohne die genaue Zusammensetzung der Strukturen kennen zu müssen.

Es werden zwei Fahrzeugtypen deklariert: Personenkraftwagen (PKW) und Lastkraftwagen (LKW):

enums.h

```
enum Fahrzeugtyp {
  TYP_PKW,
  TYP_LKW,
};

struct Auto {
  enum Fahrzeugtyp fahrzeugtyp;
  char             marke[20];
  double           maxv;
};

struct Lastwagen {
  enum Fahrzeugtyp fahrzeugtyp;
  int              achsen;
  double           gewicht;
};
```

Beide Typen haben als erstes Feld eine Typkennung, unterscheiden sich jedoch in allen restlichen Feldern. Es wird nun eine Funktion definiert, die als Argument einen Pointer auf einen dieser beiden Fahrzeugtypen enthält:

printInfo.c

```
#include "enums.h"
#include <stdio.h>

void printInfo(const void* arg) {
  switch (*(enum Fahrzeugtyp*) arg) {      // (1)

  case TYP_PKW:
    printf("Automarke %s mit %f km/h\n",
       ((struct Auto*)arg)->marke,
       ((struct Auto*)arg)->maxv);
    break;

  case TYP_LKW:
    printf("Lastwagen %d-Achser mit %ft\n",
       ((struct Lastwagen*)arg)->achsen,
       ((struct Lastwagen*)arg)->gewicht);
    break;

  }
}
```

Damit beide Typen an die Funktion übergeben werden können, wird das Argument als `void`-Pointer vereinbart.

Um innerhalb der Funktion zu unterscheiden, welcher der beiden Struktur-Typen tatsächlich vorliegt, wird das erste Feld des übergebenen Parameters in der Zeile mit dem Kommentar `(1)` abgefragt. Da bei beiden Struktur-Typen `Auto` und `Lastwagen` das Feld `fahrzeugtyp` als das erste Feld der Struktur deklariert ist, zeigt der Pointer `arg` auf eben diesen Fahrzeugtyp, egal, um welchen Struktur-Typ es sich bei dem aktuellen Funktionsaufruf handelt. So kann der Parameter `arg` einfach in einen Pointer des Typs `enum Fahrzeugtyp*` gecastet und mittels des Dereferenzierungs-Operators angesprochen werden.

Aufgrund des im ersten Feld eingetragenen Fahrzeugtyps wird zum entsprechenden `case`-Fall gesprungen und die gewünschten Informationen zu der übergebenen Struktur ausgegeben.

In der Funktion `main()` können nun beliebige Struktur-Variablen vereinbart werden und mittels des Adress-Operators an die Funktion `printInfo()` übergeben werden:

main.c

```
#include "enums.h"

void printInfo(const void* arg);

int main(void) {
  struct Auto      meinAuto   = {TYP_PKW, "Trabbi", 180.};
  struct Lastwagen meinBrummi = {TYP_LKW, 10, 40.};
  printInfo(&meinAuto);
  printInfo(&meinBrummi);
  return 0;
}
```

Die Ausgabe dieses Programmes lautet

```
Automarke Trabbi mit 180.000000 km/h
Lastwagen 10-Achser mit 40.000000t
```

Bei der Initialisierung der Variablen muss unbedingt darauf geachtet werden, dass der Fahrzeugtyp (die Kennung) korrekt gesetzt wird.

Innerhalb der Funktion `printInfo()` könnten zu Beginn auch zwei Pointer vereinbart werden. Damit würde der formale `void`-Pointer `arg` zwei lokalen Variablen zugewiesen, welche den statischen Typ `struct Auto*` und `struct Lastwagen*` besitzen. Natürlich wäre nur jeweils eine Zuweisung auch tatsächlich sinnvoll. Aufgrund dieser lokalen Variablen könnten die Elemente der übergebenen Struktur im Folgenden ganz einfach ohne ständiges Casten angesprochen werden:

```
struct Auto*      pkw = (struct Auto*)arg;
struct Lastwagen* lkw = (struct Lastwagen*)arg;
...
case TYP_PKW:
  printf("Automarke %s mit %f km/h\n", pkw->marke, pkw->maxv);
  break;
case TYP_LKW:
  printf("Lastwagen %d-Achser mit %ft\n", lkw->achsen, lkw->gewicht);
  break;
```

Es ist hierbei zu beachten, dass die Variable `lkw` im Falle `TYP_PKW` und die Variable `pkw` im Falle `TYP_LWK` jeweils nicht verwendet wird.

Aufgrund der besseren Lesbarkeit wird diese Methode häufig angewendet.

12.14 Übungsaufgaben

Aufgabe 1: Arrays in C

Geben Sie an, was das folgende Programm auf dem Bildschirm ausgibt, ohne das Programm auszuführen.

array.c

```
#include <stdio.h>

int main(void) {
  char* text = "Dichter und Denker";
  char spruch[20] = "alles weise Lenker";
  printf("\n");
  for (int i = 1; i <= 3; ++i) printf("%c", text[i]);
  printf("%c", spruch[5]);
  spruch[9] = 's';
  spruch[10] = 's';
  spruch[11] = spruch[5];
  spruch[12] = '\0';
  printf("%s", spruch + 6);
  *(spruch + 5) = '\0';
  printf("%s\n", spruch);
  return 0;
}
```

Aufgabe 2: `memcpy` und `memmove`

Studieren Sie die Kommentare in Kapitel → 12.8.2 zum Unterschied zwischen `memcpy` und `memmove`.

- Schreiben sie Ihre eigene Version von `memcpy` mittels einer einfachen `for`-Schleife.
- Testen Sie Ihre Version mit folgendem Code:

```
char string1[] = "12345678";
myMemCopy(string1 + 2, string1, strlen(string1) - 2);
```

  Das Resultat sollte sein `12121212`

- Schreiben Sie nun Ihre eigene Version von `memmove`. Testen Sie Ihre Version mit diesem Code:

```
char string2[] = "12345678";
myMemMove(string2 + 2, string2, strlen(string2) - 2);
```

  Das Resultat sollte nun lauten: `12123456`

Aufgabe 3: Pointer und Arrays bei Strings

In Kapitel → 12.7 wurde die `strcpy()`-Funktion in eigenem Code verfasst. Nebst `strcpy()` wurde jedoch auch die Funktion `strncpy()` vorgestellt. Schreiben Sie jetzt Ihre eigene Version der Funktion `strncpy()`.

Aufgabe 4: Übergabe von `char`-Arrays

Schreiben Sie eine C-Funktion `stringIndex()`, die ein bestimmtes Zeichen in einem String sucht und den Index des ersten Auftretens im String bestimmt.

Die Funktion `stringIndex()` soll zwei Übergabeparameter für den String und das Zeichen enthalten.

Wird das Zeichen nicht gefunden, soll die Funktion −1 zurückgeben.

Stellen Sie sicher, dass die Funktion const-safe ist.

Aufgabe 5: String-Verarbeitung

Implementieren Sie eine Funktion mit dem folgenden Funktionskopf:

```
void isHexadezimal(const char* string)
```

Die Funktion `isHexadezimal()` soll für jedes Zeichen eines gegebenen Strings (beispielsweise `"aBrt12"`) eine Ausgabe in der folgenden Form erstellen:

```
a: Ist eine Hex-Ziffer
B: Ist eine Hex-Ziffer
r: Ist keine Hex-Ziffer
t: Ist keine Hex-Ziffer
1: Ist eine Hex-Ziffer
2: Ist eine Hex-Ziffer
```

Aufgabe 6: Suche nach Substring

Programmieren Sie eine Funktion mit dem folgenden Prototyp:

```
int check(const char* str1, const char* str2);
```

Die Funktion `check()` gibt die erste Position in `str1` an, an der der Teilstring `str2` im String `str1` beginnt. Tritt `str2` in `str1` nicht auf, so wird `-1` zurückgegeben.

Nutzen Sie im Rahmen dieser Übung die `<string.h>`-Bibliothek nicht!

Hinweis: Verpacken Sie zu Beginn nicht alles in eine einzige Funktion. Versuchen Sie, Teilprobleme wie beispielsweise die Ermittlung der Länge eines Strings in eigene Funktionen zu verpacken.

Hinweis: Sie können die Aufgabe mit Index-Berechnung oder mit Pointer-Arithmetik lösen.