5

Intermezzo: Zeiger

Dieses Kapitel behandelt die folgenden Themen:

- Was sind Zeiger und wie benutze ich sie?
- C-Arrays und C-Strings
- Dynamisches Erzeugen von Objekten
- Parameterübergabe mit Zeigern
- Mehrdimensionale C-Arrays
- Binärdaten in Dateien schreiben und lesen
- Zeiger auf Funktionen
- Unterschied zwischen Wert- und Referenzsemantik

Zeiger sind unverzichtbare Elemente in C++. Nach der Einführung von Zeigern werden C-Arrays beschrieben und ihre Anwendung gezeigt. Verwandt mit Zeigern und C-Arrays sind C-Strings, eine andere Form von Zeichenketten. Mit Hilfe von Zeigern werden dynamisch, das heißt zur Laufzeit, erzeugte Objekte verwaltet. Auch die binäre Ein- und Ausgabe mit Dateien setzt die Begriffe Zeiger und Adresse voraus. Abschließend werden Zeiger im Zusammenhang mit Funktionen behandelt.

Viele Klassen sind in C++ ohne Zeiger nicht realisierbar. In den folgenden Kapiteln werden häufig Zeiger und C-Arrays verwendet, weswegen dieses Kapitel an dieser Stelle notwendig ist. Zeiger erlauben große Freiheiten, haben aber auch ihre Tücken.

5.1 Zeiger und Adressen

Zeiger sind ähnlich wie andere Variablen: Sie haben einen Namen und einen Wert, und sie können mit Operatoren verändert werden. Der Unterschied besteht darin, dass der Wert als *Adresse* behandelt wird. Ein Beispiel dafür sind Seitenangaben in einem Inhaltsverzeichnis, die »Adressen« für verschiedene Kapitel darstellen.

Die Namen können konventionell ein p oder ptr (für »pointer«) enthalten. Zeiger werden in C++ wie in C sehr häufig verwendet, weil sie eine große Flexibilität gestatten. Mit Hilfe von Zeigern kann dynamisch, das heißt zur Laufzeit eines Programms, Speicher beschafft werden. Anwendungen werden Sie noch kennenlernen. In Deklarationen bedeutet ein `*` »Zeiger auf«:

```
int *ip;
```

`ip` ist ein Zeiger auf einen `int`-Wert oder anders ausgedrückt: In der Speicherzelle, deren Adresse in `ip` gespeichert ist, befindet sich ein `int`-Wert. In anderen Anweisungen bedeutet `*` eine *Dereferenzierung*, das heißt, dass der Wert an der Stelle betrachtet wird, auf die der Zeiger verweist. `*ip = 100;` setzt den Wert der Speicherzelle, auf die `ip` zeigt, auf 100. Insofern könnte man die obige Deklaration lesen als: `(*ip)` ist vom Datentyp `int`.

Zeiger erhalten bei der Deklaration zunächst eine *beliebige* Adresse, genau wie andere nicht-initialisierte Variable zunächst beliebige Werte annehmen (Ausnahme: `static`-Variablen, siehe Seite 105). Daher muss vor Benutzung des Zeigers in einem Ausdruck erst eine sinnvolle Adresse zugewiesen werden, um nicht den Inhalt anderer Speicherzellen zu zerstören! Zur Verdeutlichung des Prinzips betrachten wir der Reihe nach verschiedene Deklarationen. Zunächst definieren und initialisieren wir eine Variable `i` mit der Anweisung `int i = 99;`. Wir legen damit einen Speicherplatz mit dem symbolischen Namen `i` an und tragen die Zahl 99 ein. Die uns unbekannte, vom Compiler für `i` festgelegte Speicherplatzadresse sei 10125.

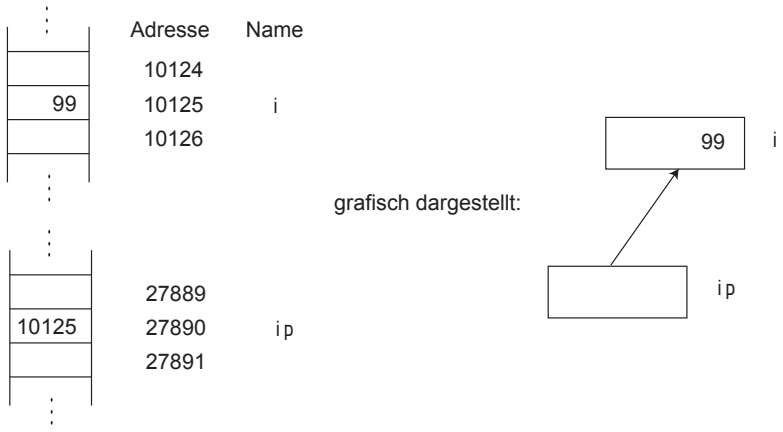

Abbildung 5.1: Zeiger

Nun werde mit `int *ip;` ein Zeiger `ip` definiert, aber nicht initialisiert. Der Wert des Zeigers ist jetzt rein zufällig. Die Speicherplatzadresse des Zeigers selbst sei 27890, und er zeigt auf eine unbekannte Adresse. Jetzt wird `ip` die Adresse von `i` zugewiesen. Dabei kommt der Operator `&` zur Anwendung, der hier als Adressoperator wirkt. In einem anderen Kontext hatten wir das Zeichen `&` bereits als Operator für die bitweise UND-Operation kennengelernt. Weisen wir nun `ip` die Adresse von `i` zu:

```
ip = &i;
```

Jetzt zeigt `ip` auf `i`. Das heißt nichts anderes, als dass die Adresse von `i`, hier 10125, bei `ip` eingetragen wird (Abbildung 5.1). Als Nächstes definieren wir einen Zeiger `ip2`, der ebenfalls auf `i` gerichtet wird. Durch die Initialisierung gleichzeitig mit der Definition durchläuft `ip2` keinen undefinierten Zustand:

```
int *ip2 = &i;
```

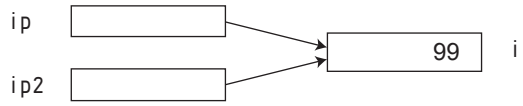

Abbildung 5.2: Zwei Zeiger zeigen auf ein Objekt.

Der Wert von `i` ist jetzt über mehrere Namen beziehungsweise Zeiger zugreifbar (Abbildung 5.2). Die Anweisungen

```
  i  = 99;
*ip  = 99;
*ip2 = 99;
```

bewirken alle dasselbe. `*ip` und `*ip2` sind *Alias-Namen* für `i`.

Hinweis zur Schreibweise von Zeigerdeklarationen

Die folgenden drei Schreibweisen sind äquivalent:

```
int* ip, x;       // 1
int * ip, x;      // 2
int *ip, x;       // 3

// Nur eine Variable pro Deklaration: Verwechslung nicht möglich
int *ip;          // 4
int x;

int* ip;          // 5
int x;
```

Die Variable `x` ist vom Typ `int`, die Variable `ip` ist vom Typ »Zeiger auf `int`«. Der Stern `*` bezieht sich also nur *auf den direkt folgenden* Namen. Um die Zuordnung beim Lesen leichter treffen zu können, sollte deshalb nur die vierte oder fünfte Schreibweise benutzt werden. Ein Zeiger auf eine Referenz ist nicht möglich.

Null-Zeiger / nullptr

In C gibt es einen speziellen Zeigerwert, nämlich *NULL*. NULL ist ein im Header `<cstddef>` definiertes Makro. Ein mit NULL initialisierter Zeiger zeigt nicht irgendwohin, sondern definitiv auf »nichts«. `NULL` ist als logischer Wert abfragbar. Um sich zu merken, dass ein Zeiger noch nicht oder nicht mehr auf ein definiertes Objekt zeigt, kann ein Zeiger auf NULL gesetzt werden: `int *iptr = NULL;`. In C++ wird NULL als Zahlenwert `0` oder `0L` (long) dargestellt. Um einen Null-Zeiger von einer Zahl unterscheiden zu können, ist im neuen C++-Standard das Schlüsselwort `nullptr` vorgesehen. Die Verwendung von NULL oder `nullptr` erhöht die Lesbarkeit an den Stellen, wo nicht unbedingt klar ist, ob ein Zeiger oder eine Zahl gemeint ist, zum Beispiel in der Parameterliste eines Funktionsaufrufs.

Typprüfung

Im Gegensatz zu C wird der Typ eines Zeigers in C++ geprüft. Gegeben seien die Definitionen:

```
char *cp;        // Zeiger auf char
void *vp;        // Zeiger auf void
```

`void` hat die Bedeutung »undefinierter Datentyp«. `vp` ist also ein Zeiger auf ein Objekt, über dessen Typ nichts gesagt wird. Eine Deklaration für ein `void`-Objekt kann es nicht geben, weil der benötigte Speicherplatz nicht angebbar ist, sehr wohl aber die Deklaration eines Zeigers, der auf ein Objekt unbekannten Typs gerichtet werden soll. Aus dem Typ des Zeigers geht der Typ des Objekts nicht hervor. Ein Zeiger auf `void` hat den Typ `void*`. Ein `void*`-Zeiger kann zum Beispiel auf ein `char`-Objekt gerichtet werden. Das Umgekehrte geht nicht.

```
vp = cp;         // möglich
cp = vp;         // nicht möglich, Fehlermeldung des Compilers!
```

Die Zuweisung kann jedoch durch eine Typumwandlung ermöglicht werden:

```
cp = static_cast<char*>(vp);
```

Solche Typumwandlungen sollte man nicht ohne wichtigen Grund benutzen, weil die Typkontrolle des Compilers umgangen wird.

Kein Zeigerzugriff auf Block-lokale Variablen!

Der Speicherplatz für die innerhalb eines Blocks deklarierten Variablen wird bei Verlassen des Blocks wieder freigegeben. Der nachträgliche Zugriff auf diese Speicherplätze kann zu Dateninkonsistenzen und zum Systemabsturz führen:

```
int i = 9;
int *ip = &i;                  // Zeiger ip zeigt auf i
*ip = 8;                       // i erhält den Wert 8
{   // neuer Block beginnt
    int j = 7;
    ip = &j;                   // Zeiger ip zeigt auf j
    // weiterer Programmcode
}                              // Blockende, j wird ungültig
*ip = 8;                       // gefährlich!
```

Die letzte Anweisung versucht, `j` über den Alias-Namen `ip` den Wert 8 zuzuweisen. Da `j` aber nicht mehr existiert, ist die Speicherstelle, auf die `ip` zeigt, möglicherweise vom Betriebssystem bereits für andere Zwecke vergeben worden. Durch diese Zuweisung werden daher gegebenenfalls andere Daten zerstört. Es gibt keine Warnung durch den Compiler oder das Laufzeitsystem! Daher sollten Sie darauf verzichten, Zeiger auf lokale Objekte mit gegenüber den Zeigern eingeschränkter Gültigkeit zu verwenden. Stattdessen sollten Sie lieber Zeiger desselben Gültigkeitsbereichs benutzen. In diesem Beispiel wäre als letzte Anweisung `j = 8;` »günstiger« gewesen, die vom Compiler sofort als falsch erkannt worden wäre und somit einen Hinweis auf Fehler im Programm gegeben hätte.

Konstante Zeiger auf konstante Werte

Die Bedeutung von `const char*` ist »Zeiger auf konstante Zeichen« und ist nicht zu verwechseln mit »konstantem Zeiger auf Zeichen« (`char *const`). Solche Deklarationen sind von rechts nach links zu lesen, wobei die Reihenfolge (`const char`) auch (`char const`) lauten kann. Ein konstanter Zeiger auf konstante Zeichen ist demnach vom Typ `const char * const` oder gleichwertig `char const * const`.

5.2 C-Arrays

Vektoren sind von Seite 81 bekannt. C-Arrays genannte Felder sind etwas Ähnliches, nur viel primitiver: Es sind Bereiche im Speicher, die (eingeschränkt) ähnlich wie ein Vektor benutzt werden können. Der Zugriff auf ein einzelnes Element eines C-Arrays geht über den von den Vektoren bekannten Indexoperator `[ ]`. Bei einer zweidimensionalen Tabelle ist zusätzlich die Spaltennummer anzugeben, zum Beispiel `[6][3]`. *AnzahlDerElemente* in Abbildung 5.3 muss eine Konstante sein oder ein Ausdruck, der ein konstantes Ergebnis hat.

Abbildung 5.3: Syntax der Definition einer eindimensionalen Tabelle

```
const int ANZAHL = 5;
int Tabelle[ANZAHL]; // Beispiel einer eindimensionalen Tabelle
```

Es ist guter Programmierstil, die Größe eines C-Arrays als *Konstante* zu deklarieren und die Konstante in der Arraydeklaration und im restlichen Programm zu verwenden. Dadurch kann ein Programm an eine andere Arraygröße angepasst werden, indem nur der Wert der Konstanten geändert wird. Es kann auch direkt eine Zahl eingetragen werden. `char a[10];` bezeichnet zum Beispiel ein Feld namens `a` mit 10 Zeichen.

Der Compiler reserviert für alle Elemente ausreichend Speicherplatz, die Anzahl der Tabellenelemente ist danach während des Programmlaufs nicht veränderbar. Die Anzahl

der Elemente ist daher problemabhängig ausreichend groß zu wählen, auch wenn einige Tabellenplätze möglicherweise nicht ausgenutzt werden. Arrays, deren Größe erst zur Laufzeit festgelegt wird, lassen sich auch konstruieren, doch davon später mehr. Die Abbildung 5.4 zeigt ein Array mit 5 ganzen Zahlen.

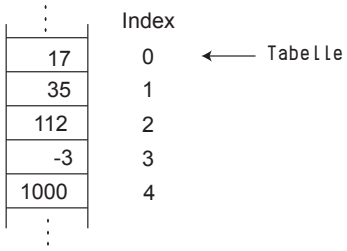

Abbildung 5.4: `int`-Array `Tabelle`

Der *Name des Feldes* zeigt auf die *Startadresse* des Feldes, d.h. auf das erste Element, und ist wie ein Zeiger einsetzbar. Anders ausgedrückt: Der Feldname ist wie ein Zeiger auf das erste Element (das heißt das Element mit dem Index 0) des Arrays. Ein Unterschied zu Zeigern besteht dennoch. Da dem Array bereits fest Speicherplatz zugewiesen ist, würde eine Änderung dieses »Zeigers« den Speicherplatz unzugänglich machen, weil die Information über die Adresse verloren geht. Daher kann ein Feldname kein sogenannter *L-Wert* sein, siehe auch Seite 953. Der Begriff L-Wert bezeichnet eine Größe, die auf der linken Seite einer Zuweisung stehen darf. Das Gegenstück ist der *R-Wert*. Also: Der Feldname ist ein R-Wert und *konstanter* Zeiger auf das erste Element des Feldes (Arrays). Der Zugriff auf ein Element ist durch den *Indexoperator* `[ ]` oder durch Zeigerarithmetik möglich, wie die Zuweisung eines Fragezeichens an das sechste Element zeigt (die Nummerierung beginnt bei 0!). Zwischen den eckigen Klammern wird die relative Tabellenposition eingetragen.

```
a[5]      = '?';    // gleichwertig:
*(a+5)    = '?';    // = Wert an der Stelle a + 5
```

Zeigerarithmetik ist hier die Addition von 5 zu einem Zeiger mit der Bedeutung, dass das Ergebnis als ein um 5 Positionen verschobener Zeiger aufgefasst wird. Einzelheiten zur Zeigerarithmetik folgen auf Seite 192.

```
char c = a[9]; // 10. Element
char* cp;     // Zeiger auf char
cp = &c;      // möglich

cp = a; // cp zeigt auf den Feldbeginn, d.h. cp == &a[0] bzw. *cp == a[0]

a = cp;       // Fehler : a ist kein L-Wert
a = &c;       // Fehler : a ist kein L-Wert
```

Zeiger und Arrays

Zeiger und Arrays sind im Gebrauch sehr ähnlich. Deswegen werden die Unterschiede hier zusammengefasst:

- Ein Zeiger ist der symbolische Name für einen Speicherplatz, der einen Wert enthält, der als Adresse benutzt werden kann.
- Ein Array (d.h. der Feldname) besitzt *in diesem Sinne keinen* Speicherplatz. Genau wie einer Konstanten keine Speicherzelle zugeordnet sein muss, weil der Compiler den Wert jedesmal direkt verwenden kann, ist keine Speicherzelle notwendig, die die Adresse des Arrays enthält. Ein Array ist vielmehr ein symbolischer Name für die Adresse (= den Anfang) eines Bereichs im Speicher. Der Name wird *syntaktisch* wie ein konstanter Zeiger behandelt.

5.2.1 C-Arrays und sizeof

C-Arrays sind als »roher Speicher« (englisch *raw memory*) *keine* Objekte im bisherigen Sinn. Es gibt keine Methoden, die verwendet werden könnten:

```
vector<double> kosten(12);
// ... berechnen
for(size_t i = 0; i < kosten.size(); ++i) // nicht bei C-Arrays!
   cout << i << ": " << kosten[i] << endl;
```

Die Größe eines C-Arrays kann nicht von ihm erfragt werden, sie muss vielmehr vorher bekannt oder mit `sizeof` ermittelt worden sein:

```
const int ANZAHL = 5;
int tabelle[ANZAHL];                    // C-Array-Definition
// ... Berechnungen
for(size_t i = 0; i < ANZAHL; ++i)
   cout << i << ": " << tabelle[i] << endl; // oder:
for(size_t i = 0; i < sizeof tabelle /sizeof tabelle[0]; ++i)
   cout << i << ": " << tabelle[i] << endl;
```

`sizeof` ist ein Operator, der den Platzbedarf eines Ausdrucks oder Typs in Bytes zurückgibt. Ein Ausdruck kann, ein Typ muss in runden Klammern eingeschlossen sein. Die Anzahl der Elemente eines C-Arrays ergibt sich einfach durch Division des Platzbedarfs für das ganze Feld durch den Platzbedarf für ein einzelnes Element, in diesem Fall das erste. Wenn der Datentyp wie hier eindeutig bekannt ist, kann statt `sizeof tabelle[0]` auch `sizeof(int)` geschrieben werden. `sizeof` funktioniert jedoch nicht bei C-Arrays, die als Parameter einer Funktion übergeben werden, weil C-Array-Parameter innerhalb der Funktion nur als Zeiger interpretiert werden. Einzelheiten sind auf Seite 211 zu finden.

5.2.2 Indexoperator bei C-Arrays

Der Zugriff über den Index-Operator `[ ]` wird nicht auf seine Grenzen überprüft! Er kann durch das entsprechende Zeigeräquivalent ersetzt werden. Man kann genausogut `*(tabelle + i)` anstatt `tabelle[i]` verwenden. Der Grund dafür liegt darin, dass der Compiler ohnehin *jeden* Zugriff über `[ ]` in die Zeigerform übersetzt, solange nicht ein benutzerdefinierter Operator `[ ]` verwendet wird. Letzterer ist nur bei Klassen möglich, siehe Kapitel 9. Mit `(tabelle+i)` ist die Adresse gemeint, die um `i` Positionen (nicht Bytes, siehe Abschnitt 5.2.4) weiter liegt als der Feldanfang, auf den `tabelle` zeigt. Der Stern '`*`' sorgt dafür, dass der Wert genommen wird, der an dieser Adresse eingetragen ist.

5.2.3 Initialisierung von C-Arrays

Ein C-Array kann bei der Definition bereits initialisiert werden. Trotz des '='-Zeichens in der Initialisierungszeile darf das Array auf der linken Seite stehen. Begründung: Eine Initialisierung ist keine Zuweisung. In C++ darf im Gegensatz zu C das '='-Zeichen entfallen. Der Compiler entnimmt die Anzahl der Feldelemente aus der Initialisierungsliste, eine Zahlenangabe kann deshalb entfallen. Die Anzahl kann mit `sizeof` ermittelt werden:

```
int feld[ ] = { 1, 777, -500};
const int ANZAHL = sizeof feld/ sizeof feld[0];
```

Falls weniger Elemente in der Initialisierungsliste als vorhanden angegeben sind, werden die restlichen Elemente mit 0 initialisiert. `int feld[3] = {1};` ist identisch mit `int feld[3] = {1, 0, 0};`.

5.2.4 Zeigerarithmetik

Wenn ein Zeiger inkrementiert oder dekrementiert wird, zeigt er nicht auf die nächste Speicheradresse, sondern auf die Adresse des nächsten Werts. Der Abstand der Speicheradressen ist gleich der Größe, die der Wert beansprucht, zum Beispiel 8 Byte bei einem Datentyp `double`. Gegeben seien folgende Deklarationen (hier gleichzeitig Definitionen):

```
double d[10];      // Array d
double *dp1 = d;
double *dp2;
```

Wenn nun `dp2 = dp1 + 1;` gesetzt wird (oder `dp2 = dp1; ++dp2;`), ergibt sich als Differenz `dp2 - dp1` der Wert 1. `dp2` zeigt also auf das nächste `double`-Element des Arrays. Aufgrund des notwendigen Platzbedarfs für eine `double`-Zahl im Speicher von angenommen 8 Byte (die Zahl mag in Ihrem System anders sein) ist die nächste `double`-Zahl also 8 Byte entfernt. Das wird ermittelt, indem man die Zeigerwerte in `long` umwandelt und dann die Differenz berechnet:

```
cout << dp2-dp1 << endl;              // ergibt 1
cout << reinterpret_cast<long>(dp2)
      - reinterpret_cast<long>(dp1) << endl;  // ergibt 8
```

Zur Wandlung des Zeigers wird der `reinterpret_cast`-Operator verwendet. Dieser Operator verzichtet im Gegensatz zum `static_cast`-Operator auf jegliche Typ-Verträglichkeitsprüfung. Nur beim Datentyp `char` wären beide Werte identisch, weil eine Variable vom Typ `char` eben genau ein Byte belegt. Mit Zeigern kann also gerechnet werden, um Adressen oder Adressendifferenzen zu ermitteln. Die Einheit ist dabei nicht Byte, sondern ergibt sich aus dem Datentyp des Zeigers.

An dieser Stelle wird die Suche eines Werts in einer Tabelle von Seite 84 aufgegriffen, nur dass jetzt ein C-Array mit Zeigeroperationen anstelle eines Vektors eingesetzt wird. Diese Variante erlaubt die kürzeste Formulierung der Schleife, setzt aber wie die 4. Variante in Abschnitt 1.9.2 voraus, dass das Feld um einen Eintrag erweitert wird, der als »Wächter« (englisch *sentinel*) dazu dient, die Schleife abzubrechen.

```
// Definitionen
const int N = ...
int a[N+1];       // C-Array
```

```
int key = ...     // gesuchtes Element
int i;            // Laufvariable
                  // Ergebnis: i = 0..N – 1 : gefunden, i = N : nicht gefunden!
```

Das (N+1). Element dient wie gesagt als »Wächter«. Der Zeiger `p` wird auf das Array gerichtet. Der Abbruch der Schleife ist durch `a[N]` als »Wächterelement« garantiert.

```
a[N] = key;
int *p = a;
while(*p++ != key);
i = p-a-1;              // Zeigerarithmetik
```

`*p++` bedeutet, erst den Wert an der Stelle `p` zu nehmen und dann `p` hochzuzählen. `*p++` ist also identisch mit `*(p++)`, wohingegen `(*p)++` den Wert an der Stelle `p` anstelle des Zeigers `p` hochzählt. Diese sehr kompakte Schreibweise trägt nicht unbedingt zum schnellen Verständnis eines Programms bei, insbesondere bei fehlenden erläuternden Kommentaren. Machen Sie sich die Wirkungsweise jedoch klar, weil viele Programmierer diese Schreibweise bevorzugen und man deren Programme schließlich auch verstehen sollte.

5.3 C-Zeichenketten

Eine *C-Zeichenkette* (englisch *string*) ist ein Spezialfall eines Arrays. Mit C-String meint man eine Folge von Zeichen des Typs `char`, die mit '`\0`' abgeschlossen wird. Diese C-Strings sind nicht zu verwechseln mit den String-Objekten von Seite 86, deren Basis sie bilden. Umgangssprachlich kann »String« sowohl ein C++-Stringobjekt wie auch einen C-String meinen. '`\0`' ist das ASCII-Zeichen mit dem Wert 0, nicht das Ziffernzeichen '0'. Der Datentyp für einen C-String ist `char*` und stellt einen Zeiger auf den Beginn der Zeichenfolge dar, über den auf den C-String zugegriffen wird. Bei der Ausgabe einer Zeichenkette »weiß« der zu `cout` gehörende Ausgabeoperator `<<`, dass `char*` *nicht* als Zeiger, sondern als mit '`\0`' terminierter String aufzufassen ist:

```
const char * str = "ABC"; // const: siehe unten
cout << str;
```

Hier wird `str` gleichzeitig definiert und initialisiert. Ein Programmierer muss an dieser Stelle '`\0`' nicht hinschreiben, weil es vom Compiler ergänzt wird.

Der Compiler erkennt einen C-String am Datentyp `char*`. Eine Zeichenkettenkonstante, auch *Literal* genannt, erkennt er daran, dass sie in Anführungszeichen eingeschlossen ist. Der vom Compiler für den String »ABC« reservierte Speicherplatz beträgt 4 Byte, ein Byte pro Zeichen und ein Byte für '`\0`'. Der Zugriff auf einzelne Zeichen ist auf die Arten möglich, die wir schon von den Arrays her kennen. `cout << str[0];` zeigt das erste Zeichen, welches an der Position 0 steht (die interne Zählung läuft auch hier ab 0). Das Zeichen mit der Nummer `i` ist ansprechbar mit `str[i]` oder `*(str+i)`. Im Gegensatz zu `char`-C-Arrays werden Stringliterale bei vielen Systemen im schreibgeschützen Speicher angelegt: Eine Änderung wie `str[0] = 'X';` führt dann zu einer Laufzeitfehlermeldung.

Deswegen: Zeiger auf Textliterale stets als `const char*` deklarieren! Eine neue Zuweisung `str = "neuer Text";` ist möglich, wobei gleichzeitig die Information über die vorherige Stelle verloren geht (siehe Abbildung 5.5).

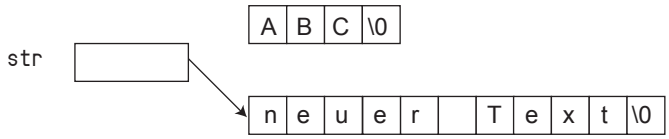

Abbildung 5.5: Neuzuweisung eines Strings

Zur Bearbeitung von C-Strings gibt es eine Menge vordefinierter Funktionen, die in der Datei *string.h* (Header `<cstring>`) deklariert sind. `#include<cstring>` veranlasst den Compiler, diese Datei zu lesen, anschließend kennt er die Funktionen. Die Funktionen sind in Abschnitt 35.9 beschrieben. Wegen der C++-Standardklasse `string` haben diese Funktionen stark an Bedeutung verloren. Hier sei beispielhaft nur die Funktion `strlen()` gezeigt, die die Anzahl der Zeichen eines Strings (ohne '`\0`') zurückgibt. Zeichen wie '`\n`' zählen als einzelne Zeichen mit, wie auch '`\"`', wodurch das Anführungszeichen als zum String zugehörig erscheint und eine vorzeitige Interpretation als Stringende verhindert wird.

Listing 5.1: Zusammengesetzter C-String

```
// cppbuch/k5/langstr.cpp
#include<cstring>
#include<iostream>
using namespace std;

int main() {
   const char* const TEXT= "Bei Initialisierung, Zuweisung, "
         "oder Ausgabe kann ein Stringliteral wie hier auch "
         "aus einzelnen Teilstrings zusammengesetzt sein.";
   cout << TEXT << endl
        << "enthält " << strlen(TEXT) << " Zeichen\n";
}
```

Der Speicherplatz für ein Literal, also eine durch Anführungszeichen begrenzte Zeichenfolge, wird zur Compilierzeit festgelegt und kann während der Ausführung des Programms nicht verändert werden. *Ein nicht initialisierter String besitzt keinen Speicherplatz außer für den Zeiger selbst* – dies wird von Anfängern häufig vergessen. Bei der Eingabe mit `cin` muss darauf geachtet werden, dass genügend Platz zur Verfügung steht, andernfalls ist das Programmverhalten unvorhersehbar.

```
// 1.
char* x;
cin >> x;  // Fehler!
```

```
// 2.
x = "12345678";
cin >> x;  // Fehler!
```

Im ersten Fall zeigt x auf eine undefinierte Stelle irgendwo im Speicher, an die die eingelesenen Zeichen geschrieben werden, wobei die vorher dort gespeicherten Informationen vernichtet werden! Im zweiten Fall wäre das Einlesen von maximal 8 Zeichen nur möglich, wenn das C++-System das Literal nicht in einem read-only-Bereich abgelegt haben sollte – dies ist normalerweise nicht der Fall. Dabei werden jedoch führende sogenannte *Zwischenraumzeichen* (englisch *whitespace*) von dem >>-Operator ignoriert. Zwischenraumzeichen ist ein Sammelbegriff für Leerzeichen, Tabulatorzeichen usw. Mit Zwischenraumzeichen wird die Eingabe für cin beendet, Einzelheiten dazu siehe in Abschnitt 2.1. Wenn wir mehr Platz bereitstellen wollen, sollte ein char-Array (siehe eine halbe Seite weiter) ausreichender Größe als Eingabepuffer deklariert werden:

```
// 3.
char bereich[1000];
cin >> bereich;  // relativ sicher, liest bis zum 1. Zwischenraumzeichen
```

Wie eine ganze Zeile inklusive aller Zwischenraumzeichen und gleichzeitig sehr sicher eingelesen wird, sehen Sie im 4. Beispiel:

```
// 4.
const int ZMAX = 100;
char zeile[ZMAX];
// sicher, liest eine Zeile, aber maximal (ZMAX - 1) Zeichen:
cin.getline(zeile, ZMAX);
```

Die Konstante ZMAX muss nur groß genug gewählt werden. Günstiger ist es in der Regel, string-Objekte einzulesen, wie auf Seite 95 beschrieben. Weitere Einzelheiten zur Methode getline() sind in Abschnitt 10.2 zu finden.

char-Arrays

Eine Art Zwitterstellung nehmen die char-Arrays ein. Genau wie bei Arrays wird Speicherplatz zur Compilierzeit reserviert. Wie Arrays können char-Arrays keine L-Werte sein. Wie bei Strings nimmt der Ausgabeoperator << bei cout an, dass mit dem char-Array eine Zeichenkette gemeint ist, die mit '\0' abschließt. Die Initialisierung kann wie bei C-Strings geschehen oder wie bei Arrays vorgenommen werden. Im Folgenden werden einige Beispiele für die verschiedenen Fälle aufgelistet.

```
// Definition und Initialisierung
char str_a [ ] = "noch ein String";
char str_b [9];           // 9 Byte reservieren, d.h. 8 Zeichen 0 bis 7 sowie '\0'
char str_c [9] = "ABC";  // 9 Byte, aber nur die ersten 4 haben definierte Werte
cin >> str_b;       // Eingabe wie bei C-Strings. Länge beachten!

// Aliasing, d.h. mehr als einen Namen für nur eine Sache
const char * str1 = "Guten Morgen!\n";
const char * str2 = str1;  // str2 zeigt auf dieselbe Stelle wie str1.
cout << str2;              // Beweis: Guten Morgen!

// erlaubte Zuweisung
char charArray1[ ] ="hallo\n";
str1 = charArray1;       // Zeiger str1 zeigt nun auf Array-Anfang
++str1;                  // str1 zeigt jetzt auf das nächste Zeichen
```

```
// nicht erlaubt, weil ein Array kein L-Wert ist:
charArray1 = str1;          // Fehler!
++charArray1;               // Fehler!

// Definition
char buchstaben[3] = "abc";       // Fehler! (kein Platz für '\0')
// tolerante Compiler ignorieren die '3'. Besser:
char buchstaben[4] = "abc";       // oder
char buchstaben[ ] = "abc";       // Compiler zählen lassen, oder
char buchstaben[3] = {'a', 'b', 'c'};  // ohne '\0'-Terminierung!
```

Beispiele: Schleifen mit Strings

Trotz vorhandener Funktionen für verschiedene Zwecke der Stringbearbeitung werden im Folgenden zur Vertiefung einige Operationen mit Strings ausführlich diskutiert, weil sie in ihrer Art typisch sind und unterschiedliche Gestaltungsmöglichkeiten für Schleifen aufzeigen. Als Erstes soll die Länge einer Zeichenkette auf verschiedene Weisen bestimmt werden. Nach jeder Version enthält `sl` die Stringlänge.

Stringlänge berechnen

Version 1

```
const char *str1 = "pro bonum, contra malum";
const char *temp = str1;
int sl = 0;
while(*temp) {
   ++sl;
   ++temp;
}
cout << "Stringlänge von " << str1 << '=' << sl << endl;
```

Die Variable `temp`, die ebenfalls auf die Zeichenfolge zeigt, ist notwendig, damit `str1` nicht geändert zu werden braucht und am Ende weiterhin auf den Beginn der Zeichenkette zeigt. Die mit 0 initialisierte Variable `sl` steht für die Stringlänge.

Was geschieht nun in der Schleife? Zunächst wird der Wert an der Stelle `temp` geprüft. Die Auswertung der Bedingung ergibt *wahr*, weil der erste Wert `*temp` mit dem ersten Zeichen 'p' der Zeichenkette identisch und damit ungleich 0 ist. `sl` wird daher um 1 erhöht, und `temp` wird auf das nächste Zeichen ('r') gerichtet. Dieser Ablauf wird so lange wiederholt, bis `*temp == '\0'` gilt, also `temp` auf das Ende des Strings zeigt. `sl` enthält jetzt die Anzahl der Zeichen in `str1`.

Version 2

```
const char * str2 = "Lieber reich und gesund als arm und krank";
const char *temp = str2;
int sl = 0;
while(*temp++)
   ++sl;
```

Die `while`-Schleife ist etwas kürzer durch den Seiteneffekt, dass `temp` nach Auswerten *in* der Bedingung inkrementiert wird. Zur Bedeutung von `*temp++` siehe die Erläuterung am Ende von Abschnitt 5.2.4.

Version 3

```
const char * str3 = "Morgenstund ist aller Laster Anfang";
int sl = 0;
while(str3[sl++]);
--sl;
```

Diese `while`-Schleife hat keine Anweisung mehr im Schleifenkörper, weil alles Nötige bereits *innerhalb der Bedingung* getan wird. Ein temporärer Zeiger ist nicht notwendig, weil über den Indexoperator auf die Elemente der Zeichenkette zugegriffen wird. Weil `sl` in jedem Fall beim Auswerten der Bedingung inkrementiert wird, also auch am Ende des Strings, muss die letzte Inkrementierung wieder rückgängig gemacht werden. Durch eine Inkrementierung noch *vor* Auswertung der Bedingung lässt sich das vermeiden:

```
int sl = -1;
while(str3[++sl]);
```

Version 4

```
const char * str4 = "letztes Beispiel zur Stringlängenberechnung";
const char *temp = str4;
int sl;
while(*temp++);
sl = temp-str4-1;
```

Diese ebenfalls sehr kurze Formulierung lässt einfach den Zeiger `temp` bis zum terminierenden Zeichen '`\0`' laufen. Die Stringlänge entspricht dann der Differenz der Zeiger, korrigiert um das '`\0`'-Zeichen.

Nachdem nun bekannt ist, wie die Länge einer C-Zeichenkette ermittelt wird, empfiehlt sich für den weiteren Gebrauch die schon aus dem Beispiel von Seite 194 bekannte Funktion `strlen(const char*)`.

Strings kopieren

Nehmen wir an, wir hätten zwei Strings deklariert, und der zweite soll ein Duplikat des ersten werden.

```
const char* original = "Ich verstehe mich! (G. Ch. Lichtenberg, geb. 1742)";
char* duplikat;
```

Die Zuweisung `duplikat = original;`, wie sie in anderen Programmiersprachen wie Pascal benutzt wird, hätte hier nicht den gewünschten Effekt, denn es wird *nur der Zeiger kopiert*, das heißt, `duplikat` zeigt auf denselben Speicherbereich wie `original`. Wenn eine C-Zeichenkette dupliziert werden soll, muss man sie *elementweise* in einen vordefinierten Speicherbereich kopieren. In diesem Fall ist keine Zieladresse definiert und nicht genügend Speicher vorhanden, weil `duplikat` bei der Definition nicht entsprechend initialisiert wurde. Es muss also zusätzlich Sorge getragen werden, dass der `duplikat` zur

Verfügung stehende Speicherplatz *mindestens* so groß ist wie der von `original`. Es gibt für Kopierzwecke ebenfalls *vordefinierte* Funktionen, auf die später noch eingegangen wird.

Das Kopieren eines Strings wird auf vier Arten mit `while` gezeigt (`do while`-Varianten sind natürlich auch möglich). Vorangestellt sind die allen Versionen gemeinsamen Definitionen, wobei sichergestellt sein muss, dass der Speicherplatz des Zielbereichs `dest` ausreichend ist.

Definitionen:

```
const char * source = "Unter allen Tieren steht der Mensch dem "
                      "Affen am nächsten.";
char dest[80];            // Platz muss reichen!
```

Version 1

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

In der Schleife wird jedem Element von `dest` das entsprechende von `source` zugewiesen. Weil die Zuweisung wegen der Schleifenbedingung nicht mehr für '`\0`' durchgeführt wird, wird die Endekennung anschließend eingetragen.

Version 2

```
int i = -1;
while(source[++i]) dest[i] = source[i];
dest[i] = '\0';
```

Dieses Beispiel unterscheidet sich vom vorhergehenden nur dadurch, dass die Inkrementierung von `i` als Seiteneffekt in die Bedingung verlegt und auf den Vergleich mit '`\0`' verzichtet wird. Es wird der Wert von `source[++i]` ausgewertet.

Version 3

```
const char *s = source;
char *d = dest;
while(*s) {*d = *s; ++s; ++d;}
*d = '\0';
```

Die Hilfszeiger `s` und `d` werden auf die Anfänge des Quell- und Zielbereichs gesetzt. In der Bedingung wird das Zeichen, auf das `s` zeigt, geprüft. Die Bedingung ist so lange wahr, bis `s` auf '`\0`' zeigt. Wenn die Bedingung wahr ist, wird jedesmal mit `*d=*s;` der Wert an der Stelle `d` gleich dem Wert an der Stelle `s` gesetzt, anschließend werden `s` und `d` um 1 weitergezählt. Auch hier wird die Zuweisung von '`\0`' nicht mehr innerhalb der Schleife vorgenommen, sodass sie nachgeholt wird. `s` wird eingeführt, damit `source` nicht verändert werden muss. Der Hilfszeiger `d` ist notwendig, weil `dest` als Array nicht veränderbar ist.

Version 4

```
const char *s = source;
char *d = dest;
while(*d++ = *s++);
```

Noch kürzer geht es nicht! Der Unterschied zur vorangehenden Version liegt darin, dass sämtliche Aktivitäten in die Bedingung verlegt werden. Alle Seiteneffekte werden ausgeführt einschließlich der Bedingung, die die Schleife zum Abbruch bringt. Die Anweisung `*d='\0';` ist nun nicht mehr gesondert notwendig, weil sie die letzte während der Bedingungsauswertung ist. Am Ende der Schleife zeigen `s` und `d` auf die Stellen direkt nach den '\0'-Zeichen.

Die sehr kompakte Schreibweise ist für Anfänger auf den ersten Blick oft schwer zu verstehen. Andererseits ist diese Art der Formulierung ein *Idiom* in C und C++, das jeder erfahrene C/C++-Programmierer kennt, weswegen man sich damit vertraut machen sollte. Die Anweisung `while(*d++=*s++)` kann als Abkürzung einer `do while`-Schleife aufgefasst werden. Zur Erläuterung wird der Ablauf dieser kurzen Anweisung in Einzelschritte aufgelöst formuliert:

```
bool x;                    // Hilfsvariable
do {
    *d = *s;               // Zuweisung eines Zeichens
    x = (*d != '\0');      // Ergebnis des Vergleichs in x merken
    ++d;
    ++s;
} while(x);
```

Nachdem nun bekannt ist, wie eine C-Zeichenkette kopiert wird, empfiehlt sich für den weiteren Gebrauch die Funktion `strcpy(char* ziel, const char* quelle)`.

C-String-Arrays

Die Elemente eines Arrays können auch C-Strings sein. Abbildung 5.6 und das Beispielprogramm verdeutlichen die Datenstruktur. Die Abbildung zeigt ein String-Array `sa` und einen Zeiger `sp` auf den Anfang des Feldes. `sa` ist ein symbolischer Name für den Feldanfang und wird syntaktisch wie ein nichtveränderbarer Zeiger auf den Feldanfang behandelt. Der Zugriff auf Feldelemente über Array-Indizes oder Zeiger ist äquivalent.

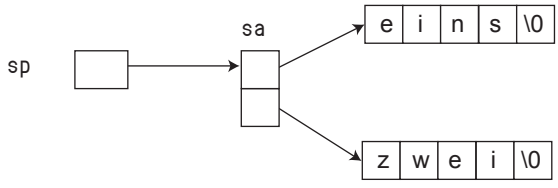

Abbildung 5.6: String-Array

Listing 5.2: C-String-Array

```
// cppbuch/k5/strarray.cpp    Bezug: Abbildung 5.6
#include<iostream>
using namespace std;

int main() {
   const char* sa[] = {"eins\n", "zwei\n"}; // Array
   const char** sp = sa; // Zeiger auf const char*.
                         // Programmausgabe:
   cout << sa[0]  << endl;   // eins
   cout << *sa    << endl;   // eins
   cout << sa[1]  << endl;   // zwei
   cout << sa[1][0] << endl; // z
   cout << *sp    << endl;   // eins
   cout << sp[1]  << endl;   // zwei
}
```

5.4 Dynamische Datenobjekte

Bisher wurden nur Datentypen behandelt, deren Speicherplatzbedarf bereits zur Compilierzeit berechnet und damit vom Compiler eingeplant werden konnte. Nicht immer ist es jedoch möglich, den Speicherplatz exakt vorherzuplanen, und es ist unökonomisch, jedesmal mit großen Arrays sicherheitshalber den maximalen Speicherplatz zu reservieren. C++ bietet daher die Möglichkeit, mit dem Operator `new` Speicherplatz genau in der richtigen Menge und zum richtigen Zeitpunkt bereitzustellen und diesen Speicherplatz mit `delete` wieder freizugeben, wenn er nicht mehr benötigt wird. Damit unterliegen die mit `new` erzeugten Objekte *nicht* den Gültigkeitsbereichsregeln für Variablen. `new` erkennt die benötigte Menge Speicher am Datentyp, sie muss also nicht explizit angegeben werden. Es gibt einen vom Betriebs- oder Laufzeitsystem verwalteten großen zusammenhängenden Adressbereich, der von [Str] *free store*, *dynamic store* oder *heap*[1] genannt wird. Ich verwende im Folgenden durchgängig die letzte Bezeichnung, die deutsch *Halde* oder *Haufen* bedeutet. Innerhalb des Heaps wird mit `new` der Platz für ein Objekt reserviert. Der Zugriff auf die neu auf dem Heap erzeugten Objekte geschieht ausschließlich über Zeiger. Abbildung 5.7 visualisiert das nachfolgende Code-Beispiel.

```
int *p;              // Zeiger auf int
p = new int;         // int-Objekt erzeugen
*p = 15;             // Wert zuweisen
cout << *p << endl;  // 15
```

Nur der Platz für `p` wird zur Compilierzeit eingeplant. Mit `p = new int;` wird Speicherplatz in der Größe von `sizeof(int)` Byte erst *zur Laufzeit* des Programms bereitgestellt, und `p`

[1] Manche unterscheiden zwischen Heap und Free Store, weil auch mit der C-Funktion `malloc` Speicher beschafft werden kann und es sein kann, dass `new` und `malloc` verschiedene Speicherbereiche nutzen. Empfehlung: Verzichten Sie auf `malloc`, wenn es keinen zwingenden Grund für die Verwendung gibt.

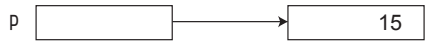

Abbildung 5.7: Erzeugen eines `int`-Datenobjekts mit `new`

zeigt nach dieser Operation auf diesen Platz. `*p` (Wert an der Stelle `p`) kann als Name für das Objekt interpretiert werden. Mit `new` kann dynamisch ein Array erzeugt werden, wobei dann eckige Klammern `[ ]` angegeben werden. Der Operator `new [ ]` zur Erzeugung von Arrays wird in C++ vom `new`-Operator für einzelne Objekte unterschieden. Hier wird ein Feld von 4 `int`-Zahlen bereitgestellt (siehe auch Abbildung 5.8).

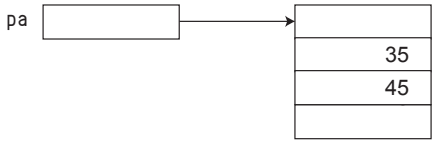

Abbildung 5.8: Erzeugen eines `int`-Arrays mit `new []`

```
// Operator new [ ]
int *pa = new int[4];    // Array von int-Zahlen
pa[1] = 35;
pa[2] = 45;
cout << pa[1] << endl; // 35
```

Der Zugriff auf die Elemente geschieht wie bei dem statisch deklarierten Array. Im Beispiel enthalten die Arrayelemente 0 und 3 undefinierte Werte. Das Programmstück zeigt beispielhaft einige Möglichkeiten des Gebrauchs von Zeigern für Objekte, die mit `new` erzeugt wurden:

```
// etwas komplizierteres Beispiel zur Übung
int **pp = new int*[4];  // Array von Zeigern auf int-Zahlen
pp[0] = p;               // pp[0] zeigt auf *p
pp[1] = &pa[2];          // pp[1] zeigt auf pa[2] (s.o.)
cout << *pp[0] << endl;  // 15
cout << **pp << endl;    // 15
cout << *pp[1] << endl;  // 45
```

Das zweifache Sternchen bei der Deklaration mag etwas ungewohnt erscheinen. `int **pp` ist gleichbedeutend mit `(int*)* pp`, das heißt, `pp` ist ein »Zeiger auf Zeiger auf `int`«, weil die Array-Elemente selbst Zeiger sind. Sie sind nach der Deklaration alle noch undefiniert, verweisen also nicht auf bestimmte Speicherplätze. Das ändert sich nur für die ersten beiden Elemente, die nach den Zuweisungen auf die oben definierten Plätze `*p` und `pa[2]` verweisen (siehe Abbildung 5.9).

Dynamisch erzeugte Struktur

Die Struktur `test_struct` enthält eine `int`- und eine `double`-Variable und außerdem einen Zeiger `next` auf ein gleichartiges Objekt.

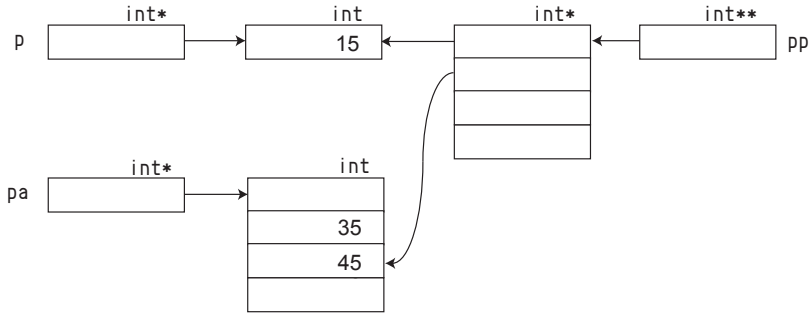

Abbildung 5.9: Array von `int`-Zeigern zum Programmbeispiel

```
struct test_struct {
    int a;
    double b;
    test_struct* next;
};

test_struct *sp= new test_struct;
```

Der Zeiger `sp` verweist auf ein neu erzeugtes Objekt von der Größe der Struktur `sizeof(test_struct)`, der Summe `sizeof(int) + sizeof(double) + sizeof(test_struct*)` entsprechend. Der Zugriff auf die Elemente der Struktur geschieht über den Pfeil-Operator `->`:

```
sp->a = 12;        // entspricht (*sp).a
sp->b = 17.4;
```

Dem Zeiger `sp->next` wird mit `sp->next = new test_struct;` ein weiteres neu erzeugtes Objekt zugewiesen, sodass jetzt zwei miteinander verkettete Objekte vorliegen, die beide über den Zeiger `sp` erreichbar sind (Abbildung 5.10).

```
// weiteres Objekt erzeugen
sp->next = new test_struct;
// Zugriff auf Element a des neuen Objekts
sp->next->a = 144;
```

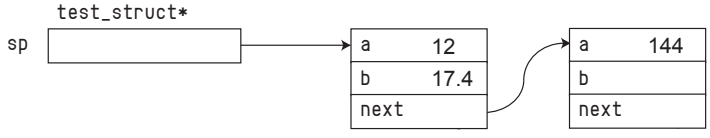

Abbildung 5.10: Struktur mit Verweis auf zweite Struktur

Interne Datenelemente des zweiten Objekts können durch entsprechende Anweisungen der Art `sp->next->a = 144;` erreicht werden. Der beschriebene Verkettungsmechanismus wird zum Aufbau von Listen verwendet.

5.4.1 Freigeben dynamischer Objekte

Der `delete`-Operator gibt den reservierten Platz wieder frei, damit er von Neuem belegt oder anderen Programmen zur Verfügung gestellt werden kann. Nach `delete` wird der Zeiger angegeben, der auf das zu löschende Objekt verweist. Im Folgenden werden alle oben erzeugten Objekte der Reihe nach gelöscht. Dabei könnte man anstatt `delete p` auch `delete pp[0]` schreiben, weil `pp[0]` auf dasselbe Objekt verweist, aber nicht beides. Das Löschen des Objekts, auf das `sp->next` zeigt, muss vor dem Löschen von `*sp` erfolgen, weil sonst mit der Vernichtung von `*sp` die Information über den Ort des über `next` verketteten Objekts verloren wäre.

In Entsprechung zu den Operatoren `new` und `new [ ]` wird zwischen den Operatoren `delete` und `delete [ ]` unterschieden.

```
// Freigaben zu Abbildung 5.9
delete p;
delete [ ] pa;              // Array löschen
delete [ ] pp;              // Array löschen
```

```
// Freigaben zu Abbildung 5.10. Reihenfolge beachten!
delete sp->next;                    // muss zuerst kommen
delete sp;
```

Falls es nicht gelingt, mit `new` Speicher zu beschaffen, zum Beispiel weil schon zu viel verbraucht und nicht wieder freigegeben wurde, wird das Programm mit einer Fehlermeldung abgebrochen (Einzelheiten siehe in Abschnitt 8.2).

Einige Dinge sollten bei der Verwendung von `new` und `delete` beachtet werden, deren Missachtung oder Unkenntnis in manchen Fällen ein unvorhersehbares Programmverhalten nach sich zieht, leider *ohne* Fehlermeldung seitens des Compilers oder des Laufzeitsystems.

- `delete` darf *ausschließlich* auf Objekte angewendet werden, die mit `new` erzeugt worden sind.

  ```
  int i;
  int *iptr = &i;
  delete iptr;       // Fehler! (Absturzgefahr)
  iptr = new int;
  delete iptr;       // ok!
  ```

- `delete` darf nur *einmal* auf ein Objekt angewendet werden. Der Wert eines Zeigers, auf den `delete` angewendet wurde, ist danach *undefiniert*, er ist also leider nicht gleich NULL bzw. 0. Falls zwei oder mehr Zeiger auf ein Heap-Objekt zeigen, bewirkt ein `delete` auf nur einen von diesen Zeigern die Zerstörung des Objekts. Die anderen Zeiger verweisen danach nicht mehr auf einen Speicherbereich mit Bedeutung, sie heißen dann »hängende Zeiger« (englisch *dangling pointer*).
- `delete` auf einen Null-Zeiger angewendet, bewirkt nichts und ist unschädlich.
- Wenn ein mit `new` erzeugtes Objekt mit dem `delete`-Operator gelöscht wird, wird automatisch der Destruktor für das Objekt aufgerufen.
- Mit `new` erzeugte *Objekte* unterliegen *nicht* den Gültigkeitsbereichsregeln für Variablen. Sie existieren so lange, bis sie mit `delete` gelöscht werden oder das Programm

beendet wird, unabhängig von irgendwelchen Blockgrenzen. Dies gilt nicht für die statisch deklarierten *Zeiger* auf diese Objekte, für die die normalen Gültigkeitsbereichsregeln gelten. Als Konsequenz ist darauf zu achten, dass ein Zeiger auf ein Heap-Objekt mindestens bis zu dessen Löschung existiert, damit überhaupt eine Chance besteht, das Objekt zu löschen. Das folgende Programmfragment verdeutlicht das Problem:

```
{ // Blockanfang
   // ungünstig
   int *p = new int;
   *p = 30000;
   // mehr Programmcode
}
```

Nach Verlassen des Blocks existiert `p` nicht mehr; das Objekt mit dem ehemaligen Namen `*p` existiert noch, ist aber nicht mehr erreichbar. In Schleifen angewendet, kann diese Methode schnell zur Speicherknappheit führen. Es kommt vor, dass ein rund um die Uhr laufendes Programm nach einer Woche plötzlich ohne erkennbare Ursache stehenbleibt, wofür wir nun einen möglichen Grund kennen. Ein nicht mehr zugänglicher Bereich wird »verwitwetes« Objekt genannt, im Englischen »Speicherleck« *memory leak*. Richtig ist:

```
{
   int *p = new int;
   *p = 30000;
   // mehr Programmcode
   delete p;                  // *p wird nicht mehr gebraucht
}
```

oder:

```
int *p;                       // außerhalb des Blocks deklariert
{
   p  = new int;
   *p = 30000;
   // mehr Programmcode
}
cout << *p;                   // weitere Verwendung von p
// mehr Programmcode
delete p;
```

- Die Freigabe von Arrays erfordert die Angabe der eckigen Klammern (siehe obiges Beispiel). Ohne `[ ]` gäbe `delete pa;` nur ein einziges Element frei! Der Rest des Arrays wäre danach nicht mehr zugänglich (verwitwetes Objekt). *Merkregel:* `delete [ ]` dann (und *nur* dann) benutzen, wenn die Objekte mit `new [ ]` erzeugt worden sind.

Hinweis

Wenn statt `delete [ ]` nur `delete` geschrieben wird, sich aber aus dem Kontext ergibt, dass es um die Löschung eines Arrays gehen soll, wird von manchen Compilern die fehlerhafte Schreibweise stillschweigend korrigiert. Nach dem C++-Standard resultiert

eine fehlerhafte `delete`-Anweisung für ein Array (d.h.ohne die eckigen Klammern) aber in »undefiniertem Verhalten« (englisch *undefined behaviour*).

Das bedeutet nicht unbedingt Programmabsturz, sondern dass der Hersteller des Compilers selbst entscheiden kann, was zu tun ist: Fehler selbsttätig korrigieren, Warnung ausgeben, Speicherleck akzeptieren oder was auch immer. Das bedeutet aber auch, dass ein Programm der Art

```
while(true) {                  // Programm soll ständig laufen
   int* arr = new int[100000];
   // ... mit dem Array arbeiten
   delete arr;  // Klammern [ ] fehlen!
}
```

nicht portabel ist! Bei einem System wird der Fehler toleriert, beim anderen kracht es, weil der Speicher ausgeht. Beides wäre im Sinn des C++-Standards »legal«. Manche Programmiersprachen, wie etwa Java, kennen kein `delete`. Dafür gibt es ein parallel laufendes Programm zur Speicherbereinigung, *garbage collector* genannt (englisch für: »Müllsammler«). Die Speicherverwaltung vieler C++-Systeme enthält aus Effizienzgründen keine Speicherbereinigung – es gibt ja `delete`. C++ erlaubt es jedoch, eine eigene Speicherverwaltung mit diesen und anderen Eigenschaften zu schreiben. Es gibt auch entsprechende Bibliotheken – ein einfacherer Weg, die Möglichkeit zur garbage collection zu erhalten.

Neben mangelndem Speicher liegt ein weiterer Grund für das plötzliche unerwartete Stehenbleiben von sehr lange laufenden Programmen in der Zerstückelung (Fragmentierung) des Speichers durch viele `new`- und `delete`-Operationen. Damit ist gemeint, dass sich kleinere belegte und freie Plätze abwechseln, sodass die plötzliche Anforderung eines größeren *zusammenhängenden* Bereichs für ein großes Datenobjekt nicht mehr erfüllbar ist, obwohl die Summe aller einzelnen freien Plätze ausreichen würde. Dieses Problem verlangt ein Zusammenschieben aller belegten Plätze, sodass ein großer freier Bereich entsteht. Programme zur garbage collection erledigen diese Aufgabe meistens gleich mit.

5.5 Zeiger und Funktionen

5.5.1 Parameterübergabe mit Zeigern

Wenn ein übergebenes Objekt modifiziert werden soll, kann die Übergabe *per Zeiger* auf das Objekt geschehen. Dies ist ein Spezialfall der Übergabe per Wert, es wird nämlich mit einer *Kopie* des Zeigers weitergearbeitet, die auf dasselbe Objekt wie das Original zeigt. Die Übergabe per Zeiger ist kein Sprachmittel von C++, sondern zeigt nur eine andere Art der Benutzung der Übergabe per Wert. Eine Modifikation des Zeigers in der Funktion ändert also nicht den Wert des Zeigers für den Aufrufer.

Das Objekt, auf das der Zeiger verweist, kann in der Funktion gleichwohl geändert werden, sodass der Zeiger beim Aufrufer zwar auf dieselbe Adresse zeigt, unter dieser Adresse

jedoch ein verändertes Objekt zu finden ist. Im Beispiel ist beides zu sehen: Die lokale Kopie `s` des Zeigers `str` wird verändert, ohne dass `str` verändert wird, aber das Objekt an der Stelle `str` ändert sich, weil alle Klein- durch Großbuchstaben ersetzt werden. Abbildung 5.11 zeigt in Ergänzung zum Programmbeispiel, dass ein Objekt über die Kopie eines Zeigers zugreifbar und modifizierbar ist.

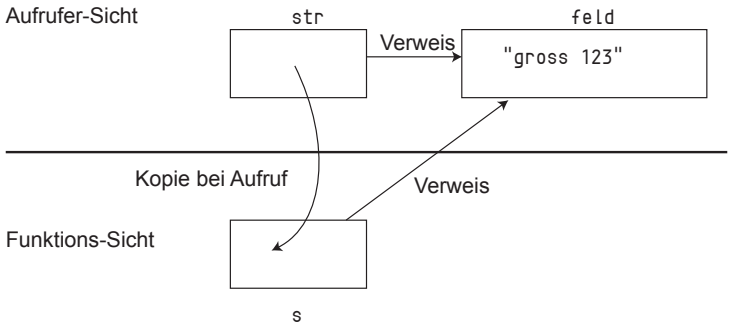

Abbildung 5.11: Parameterübergabe per Zeiger (Bezug: Beispielprogramm)

Listing 5.3: Parameterübergabe per Zeiger

```
// cppbuch/k5/perzeigISO8859.cpp
#include<iostream>
using namespace std;
void upcase(char *);   // Prototyp

int main( ) {
   char feld[] = "gross 123"; // Array
   char* str = feld;          // Zeiger
   upcase(str);               // upcase(feld); ist ebenso möglich
   cout << str << endl;       // GROSS 123
}

void upcase(char* s) {
   const int DIFFERENZ = 'a' - 'A'; // In der ASCII-Tabelle sind die Platznummern
   // der Kleinbuchstaben um 'a'-'A' = 32 gegenüber den Großbuchstaben verschoben.
   while(*s) {
      if(*s>='a' && *s<='z') {
         *s -= DIFFERENZ;
      }
      else { // Umlaute
         // Lesbar, aber nicht auf jedem System: Umlaute sind nicht portabel! Der
         // folgende Teil funktioniert nur dann richtig, wenn dieses Programm vor der
         // Compilation dieselbe Zeichenkodierung hat wie das Terminal und die
         // Umlaute als ein Byte darstellbar sind, wie etwa bei der Zeichenkodierung
         // ISO8859-1. Weitere Informationen finden Sie im Abschnitt 31.
         switch(*s) {
            case 'ä' : *s = 'Ä'; break;
```

```
            case 'ö' : *s = 'Ö'; break;
            case 'ü' : *s = 'Ü'; break;
            default:; // nichts tun
         }
      }
      s++;
   }
}
```

Weil ein `char` vom Compiler zwecks Umwandlung wie eine Ein-Byte-`int`-Zahl interpretiert wird, ist das Rechnen ohne explizite Typumwandlung möglich. Um von der internen Darstellung der Zeichen unabhängig zu sein, wird `'a'-'A'` anstatt 32 benutzt. Auch ist es schwierig für einen Leser, bei fehlendem Kommentar die Bedeutung der Zahl 32 zu erraten. Die Umlaute und andere Sonderzeichen werden auf verschiedenen Maschinen mit anderen Betriebssystemen durch andere Codierungen repräsentiert, darunter auch Multi-Byte-Codierungen wie Unicode. Alle Zeichen außerhalb des ASCII-Bereichs sind nicht portabel und müssen ihrer Codierung entsprechend verarbeitet werden.

const und Zeiger-Parameter

Hier seien verschiedene Möglichkeiten der Verwendung von `const` in einer Parameterliste aufgezeigt (vergleiche Seite 189). Es liegt ein C-Array `int tabelle[100]` zugrunde.

- `void tabellenfunktion(int* tabelle)`
  Nichts ist konstant, das heißt, in der Funktion können sowohl die Tabellenwerte wie auch der Zeiger `tabelle` geändert werden. Ersteres wirkt sich beim Aufrufer aus, Letzteres nicht, weil in der Funktion eine Kopie des Zeigers angelegt wird (Übergabe des Zeigers per Wert).
- `void tabellenfunktion(const int* tabelle)`
  `tabelle` zeigt auf nicht veränderbare Elemente. Eine Anweisung etwa der Art `tabelle[0] = 3;` innerhalb der Funktion würde vom Compiler nicht akzeptiert werden, `++tabelle` wäre hingegen ok.
- `void tabellenfunktion(int* const tabelle)`
  `tabelle` ist selbst konstant. Eine Anweisung `tabelle = NULL;` würde vom Compiler nicht akzeptiert werden, `tabelle[10] = 17;` schon.
- `void tabellenfunktion(const int* const tabelle)`
  kombiniert die beiden vorhergehenden Möglichkeiten: Ein konstanter Zeiger zeigt auf unveränderliche Werte.

Ein konstanter Zeiger verbietet eine Änderung des Zeigers innerhalb der Funktion. Diese Änderung hätte aber ohnehin keine Auswirkung auf das aufrufende Programm, wie oben erläutert. In der Praxis sind daher nur die ersten beiden Fälle von Bedeutung.

5.5.2 Parameter des main-Programms

`main()` kann über Parameter verfügen, siehe Seite 116. Sie sind innerhalb des Programms auswertbar, wie es in vielen Dienstprogrammen gehandhabt wird. Die Parameter werden beim Aufruf des Programms auf Betriebssystemebene nach dem Programmnamen angegeben, daher der Name »Kommandozeilenparameter«. Der Compilerhersteller kann

weitere Parameter nach `char* argv[]` zulassen, üblich ist ein weiteres String-Array `char* env[]` zum Abfragen von Umgebungsvariablen (englisch *environment variables*):

```
int main( int argc, char* argv[], char* env[]) {
 // ...
}
```

`argc` ist die Anzahl der Kommandozeilenparameter einschließlich des Programmaufrufs, `argv[]` ein String-Array mit den Kommandozeilenparametern. Das letzte Element von `env[]` ist 0, und es gilt `argv[argc] == 0` (Null-Pointer). `argv[0]` enthält den Programmaufruf. Das Programm *mainpar.cpp* demonstriert die Benutzung der Parameter. Auf Betriebssystemebene kann es zum Beispiel mit *mainpar 1 par2 /3 5 5 A6* aufgerufen werden. Die Parameter haben hier keine Bedeutung und dienen nur zur Demonstration.

Listing 5.4: Kommandozeilenparameter anzeigen

```
// cppbuch/k5/mainpar.cpp
#include<iostream>
using namespace std;

int main(int argc, char* argv[], char *env[]) {
    cout << "Aufruf des Programms = " << argv[0] << endl;
    cout << (argc-1) << " weitere Argumente wurden main() übergeben:\n";
    int i = 1;
    while(argv[i]) {
         cout << argv[i++] << endl;
    }
    cout << "\n*** Umgebungs-Variablen: ***\n";
    i = 0;
    while(env[i]) {
         cout << env[i++] << endl;
    }
}
```

5.5.3 Gefahren bei der Rückgabe von Zeigern

Wie bei der Rückgabe von Referenzen (Seite 112) muss auch bei Zeigern darauf geachtet werden, dass sie nicht auf lokale Objekte verweisen, die nach dem Funktionsaufruf verschwunden sind.

Negativ-Beispiel

```
#include<iostream>

char* murks(const char * text) {
  char neu[100];           // Speicherplatz besorgen
  char* n = neu;
  while(*n++ = *text++);   // text wird nach neu kopiert
  return neu;              // Fehler!
}
```

```
int main() {
  char *sp3 = murks("Oh je!");
  std::cout << sp3;        // nicht existierendes Objekt!
}
```

Alternative

Die Funktion `murks()` soll ein Duplikat des übergebenen Strings erzeugen und den Zeiger darauf zurückgeben. Der Speicherplatz für das lokale Feld `neu` wird jedoch bei Verlassen der Funktion freigegeben! Die Kopie wird zerstört. Richtig wäre es, den Speicherplatz mit `new` zu besorgen und den Zeiger `neu` zurückzugeben:

```
char* kein_murks(const char * text) {
  char *neu = new char[strlen(text) + 1]; // strlen() erfordert #include<cstring>
  char* n = neu;
  while(*n++ = *text++);            // text wird nach neu kopiert
  return neu;
} // Der Aufrufer muss für die Speicherfreigabe sorgen!
```

Ferner soll kein Objekt per `return` zurückgegeben werden, das mit dem `new`-Operator in einer Funktion erzeugt wurde. Der Grund liegt darin, dass bei der Rückgabe mit `return` ein Objekt *kopiert* wird. Das Original wäre für ein notwendiges `delete` nicht mehr erreichbar. Es darf also nur der Zeiger auf ein mit `new` erzeugtes Objekt zurückgegeben werden, wobei der Aufrufer die Verantwortung hat, das Objekt irgendwann zu löschen.

5.6 this-Zeiger

`this` ist ein Schlüsselwort, das innerhalb einer Elementfunktion einen *Zeiger auf das Objekt* darstellt. Weil `this` der Zeiger auf das Objekt ist, wird das Objekt selbst durch `*this` benannt, ganz in Analogie zu den Zeigern, die wir schon kennen: `*ptr = 3;` weist den Wert 3 dem Objekt zu, auf das `ptr` zeigt. Innerhalb einer Methode bezeichnet `this` einen Zeiger auf das aktuelle Objekt und `*this` das Objekt selbst, für das die Methode aufgerufen wird. `*this` ist nur ein anderer Name (Alias-Name) für das Objekt, der innerhalb der Elementfunktionen benutzt werden kann.

5.7 Mehrdimensionale C-Arrays

5.7.1 Statische mehrdimensionale C-Arrays

Gelegentlich hat man es mit mehrdimensionalen Feldern zu tun, am häufigsten mit zweidimensionalen. Eine zweidimensionale Tabelle besteht aus Zeilen und Spalten, eine dreidimensionale aus mehreren zweidimensionalen Tabellen. In C++ wird eine Tabelle linear

auf den Speicher abgebildet. Im Fall der zweidimensionalen Tabelle kommen zunächst alle Elemente der ersten Zeile, dann alle Elemente der zweiten Zeile usw. Ein zweidimensionales Array ist ein Array von Arrays. Das folgende Beispiel zeigt eine statisch angelegte zweidimensionalen Tabelle.

Listing 5.5: Zweidimensionale Matrix

```
// cppbuch/k5/matrix2d.cpp
#include<iostream>
using namespace std;

int main() {
   const size_t DIM1 = 2;
   const size_t DIM2 = 3;
   int matrix [DIM1] [DIM2] = { {1,2,3}, {4,5,6} };
   for (size_t i = 0; i < DIM1; ++i) {
      for (size_t j = 0; j < DIM2; ++j) {
         cout << matrix[i][j] << ' ';
      }
      cout << endl;
   }
}
```

Die Matrix besteht aus zwei Zeilen und drei Spalten. Die Struktur wird auch in der Initialisierungsliste deutlich. Das Weglassen der inneren geschweiften Klammern, das heißt `{1, 2, 3, 4, 5, 6}`, hätte die gleiche Wirkung. Auch hier gilt, dass nicht aufgeführte Elemente der Liste mit 0 initialisiert werden. `{{1}, {4}}` ist gleichbedeutend mit `{{1, 0, 0}, {4, 0, 0}}`, also Initialisierung der ersten Spalte und Nullsetzen des Rests.

Die Konstanten `DIM1` und `DIM2` müssen zur Compilationszeit bekannt sein. In Abschnitt 5.7.2 werden Sie sehen, wie die Arraygröße erst zur Laufzeit des Programms definiert werden kann. Mehrdimensionale Arrays haben für *jede Dimension* ein Klammernpaar `[]`.

In C++ darf der Zugriff auf ein Element eines mehrdimensionalen Arrays nicht mit einem Komma abgekürzt werden, wie es in anderen Programmiersprachen manchmal der Fall ist. Es darf nicht `a[2, 3]` statt `a[2][3]` geschrieben werden. Der Compiler meldet die falsche Schreibweise *nicht* unbedingt, weil das Komma einen besonderen Operator darstellt und der aus semantischer Sicht falsche Ausdruck syntaktisch erlaubt sein kann.

Der Kommaoperator gibt eine Reihenfolge von links nach rechts vor. Der Ausdruck `sum = (total = 3) + (++total)` von Seite 57, in dem die Auswertungsreihenfolge der Klammerausdrücke nicht definiert ist, kann mit dem Kommaoperator in einen definierten Ausdruck verwandelt werden: `sum = (total = 3, total + ++total)`. Das Ergebnis des gesamten Ausdrucks ist das Ergebnis des Teilausdrucks nach dem letzten Komma. Im obigen Fall würde fälschlicherweise `a[2, 3]` als `a[3]` interpretiert werden.

Tipp

Große Arrays sollten *dynamisch*, wie in Abschnitt 5.7.2 beschrieben, angelegt werden, weil auf dem Heap im Allgemeinen mehr Platz als auf dem Stack ist.

Array als Funktionsparameter

Wie schon in Abschnitt 5.2 erwähnt, sind C-Arrays syntaktisch mit konstanten Zeigern gleichzusetzen. Die Parameterliste `(int tabelle[])` einer Funktion ist daher dasselbe wie `(int* tabelle)`. Damit kann `sizeof` nicht zur Größenermittlung eines Arrays benutzt werden, wie das folgende Beispiel zeigt:

```
// fehlerhaftes Beispiel
void tabellenausgabe(int tabelle[]) { // C-Array-Deklaration
   int anzahlBytes = sizeof tabelle; // Fehler! Grund: dasselbe wie sizeof(int*)!
   int wieoft = anzahlBytes/sizeof(int);
   for(int i = 0; i < wieoft; ++i) {
      cout << tabelle[i] << endl;
   }
}

int main() {
   const int ANZAHL = 5;
   int tabelle[ANZAHL];          // C-Array-Definition
   // ... Berechnungen
   tabellenausgabe(tabelle);     // leider falsch ...
}
```

`sizeof tabelle` innerhalb der Funktion `tabellenausgabe` gibt nur den Platzbedarf für einen Zeiger zurück, weil syntaktisch `int tabelle[]` dasselbe wie `int *tabelle` ist! `sizeof` kann den benötigten Speicherplatz nur im Sichtbarkeitsbereich der *Definition* eines C-Arrays erkennen, also dort, wo Speicherplatz tatsächlich beschafft wird (wie etwa auf Seite 191). Jede andere Stelle, wo ein C-Arrayname eingeführt wird, ist eine *Deklaration* ohne gleichzeitige Definition (vergleiche die »one definition rule« auf Seite 127).

Merke:
Die Anzahl der Array-Elemente muss einer Funktion übergeben werden! Dies gilt auch für mehrdimensionale Arrays.

```
// korrigiertes Beispiel
void tabellenausgabe1D(int tabelle[], size_t n) {
   for(size_t i = 0; i < n; ++i) {
      cout << tabelle[i] << endl;
   }
}
```

Wie sieht es bei einem zwei- oder mehrdimensionalen Array aus? Das ist ein Array von Arrays, das heißt, hier gibt es eine zu übergebende Anzahl von Arrays (statt Werten wie im eindimensionalen Fall). Wie erfährt die Funktion von der Größe der Arrays? Bei statischen, also nicht mit `new` erzeugten Arrays, geht die Größeninformation mit in den Typ ein (zu den dynamischen Arrays siehe unten). Beispiel:

```
int feld1[2][3] = {{1,2,3},{4,5,6}};
```

ist ein Array, das 2 Elemente enthält, nämlich zwei Arrays zu je 3 `int`-Werten (2 Zeilen, 3 Spalten). Der Typ der zuletzt genannten Arrays ist *Bestandteil des Arraytyps* von `feld1`. Dies gilt entsprechend für Zeiger:

```
// kompatibler Zeiger
int (*pFeld)[3] = feld1; // zeigt auf Zeile feld1[0]
```

Die Funktion zur Ausgabe dieses Feldes benötigt diese Typinformation in der Parameterliste:

```
void tabellenausgabe2D(int (*T)[3], size_t n) {
   // alternativ:   void tabellenausgabe(int T[][3], size_t n)
  for(size_t i = 0; i < n; ++i) {
    for(size_t j = 0; j < 3; ++j) {
      cout << T[i][j] << ' ';
    }
    cout << endl;
  }
  cout << endl;
}
```

Mit dieser Typinformation weiß der Compiler, wo jede Zeile anfängt. Innerhalb der Funktion ist `sizeof T[0]` gleich 3 mal `sizeof(int)`. Die Funktion kann für das Array oder den Zeiger gleichermaßen aufgerufen werden:

```
tabellenausgabe2D(feld1, 2); // 1 2 3
                             // 4 5 6
++pFeld;                     // zeigt jetzt auf Zeile feld1[1]
tabellenausgabe2D(pfeld, 1); // 4 5 6
```

Dieses Schema setzt sich für mehrere Dimensionen fort. Die Information `[3][4]` gehört zum Typ einer dreidimensionalen Matrix `int Matrix3D[2][3][4]`. Die Funktion zur Ausgabe dieser Matrix hat die Schnittstelle

```
void tabellenausgabe3D(int (*T)[3][4], size_t n) ; // oder
void tabellenausgabe3D(int T[][3][4], size_t n) ;
```

Wie schreibt man eine Funktion, die für statische zwei-dimensionale Arrays geeignet ist, egal wie groß die Spaltenanzahl ist? Die Lösung ist ein Template. Der Compiler leitet dann den Feldtyp bei Aufruf der Funktion aus dem Parameter ab.

```
template<typename Feldtyp>
void tabellenausgabe2D(Feldtyp tabelle, size_t n) {
  const size_t SPALTEN = sizeof tabelle[0] /sizeof tabelle[0][0];
  for(size_t i = 0; i < n;++i) {
    for(size_t j = 0; j < SPALTEN;++j) {
      cout << tabelle[i][j] << ' ';
    }
    cout << endl;
  }
  cout << endl;
}
```

Interpretation von [ ], [ ][ ] usw.

Der Compiler wandelt die Indexoperatoren mehrdimensionaler Arrays wie bei den eindimensionalen Arrays in die Zeigerdarstellung um. Wenn wir mit `X` alles bezeichnen, was

vor dem letzten Klammernpaar steht, und mit `Y` den Inhalt des letzten Klammerpaares, so wird vom Compiler `X[Y]` in `*((X)+(Y))` umgesetzt. Auf `X` wird das Verfahren wiederum angewendet, bis alle Indexoperatoren aufgelöst sind. Man kann daher statt `matrix[i][j]` ebenso `*(matrix[i]+j)` oder `*(*(matrix+i)+j)` schreiben. `matrix[i]` ist als Zeiger auf den Beginn der `i`-ten Zeile zu interpretieren. Durch die Zeigerarithmetik wird die dahinterstehende Berechnung der tatsächlichen Adresse verborgen, die ja noch die Größe der Datenelemente eines Arrays berücksichtigen muss. Die Position `(matrix + i)` liegt daher (`i` mal `sizeof(matrix[0])`) Bytes von der Stelle `matrix` entfernt. Dies wird bei der Ermittlung der Anzahl der Spalten im obigen Funktions-Template ausgenutzt.

Übungen

5.1 Auf Seite 191 wird über die Äquivalenz von `*(kosten+i)` und `kosten[i]` gesprochen. Anstatt `(kosten+i)` könnte man genausogut `(i+kosten)` schreiben, das Ergebnis der Addition wäre das gleiche. Ist es dann richtig, dass die Schreibweise `i[kosten]` äquivalent ist zu `kosten[i]`?

5.2 Geben Sie den für `matrix[2][3]` benötigten Speicherplatz in Bytes an, wenn `sizeof(int)` als 4 angenommen wird. An welcher Bytenummer beginnt das Element `matrix[i][j]` relativ zum Beginn das Arrays?

5.3 Schreiben Sie die Multiplikation zweier Matrizen `a[n][m] * b[p][q]`. Das Ergebnis soll in einer Matrix `c[r][s]` stehen. Welche Voraussetzungen gelten für die Zeilen- und Spaltenzahlen n, m, p, q, r, s?

5.4 Schreiben Sie zur Ausgabe von dreidimensionalen Arrays eine Template-Funktion `tabellenausgabe3D(Feldtyp T, size_t n)` entsprechend dem obigen Muster für zwei Dimensionen.

5.7.2 Dynamisch erzeugte mehrdimensionale Arrays

Mehrdimensionale Arrays mit konstanter Feldgröße

Auf Seite 209 haben Sie die statische Deklaration von mehrdimensionalen Feldern gesehen. Im Abschnitt 5.4 wurden ein Array von `int`-Zahlen und ein Array von Zeigern auf `int` dynamisch erzeugt, also zur Laufzeit des Programms.

Wie erzeugt man nun dynamisch mehrdimensionale Arrays, zum Beispiel eine zwei- oder dreidimensionale Matrix? Betrachten wir die Anweisung `int *pa = new int[4];`, die eine Zeile aus `int`-Elementen anlegt, so können wir feststellen, dass `new` einen Zeiger vom Datentyp »Zeiger auf Typ eines Arrayelements« zurückgibt. Eine zweidimensionale Matrix ist aber nichts anderes als ein Array von Arrays, wie oben beschrieben. Ein Element dieses Arrays ist eine Zeile mit zum Beispiel 7 Elementen, und daher sollte `new` einen Zeiger vom Datentyp »Zeiger auf eine Zeile mit 7 Elementen« zurückgeben:

```
int (* const p2)[7]= new int [5][7];
```

`p2` ist ein konstanter Zeiger, der auf die Zeile 0 einer zweidimensionalen Matrix mit 5 Zeilen zu je 7 `int`-Zahlen verweist. Der Zugriff auf ein einzelnes Element kann anschließend mit einer Anweisung wie zum Beispiel `p2[1][6] = 66;` erreicht werden. Man beachte, dass die »5« nach `new` auch eine andere Zahl sein kann – die auf beiden Seiten auftretende »7« jedoch nicht, weil sie in den *Datentyp* eingeht. Der Zeiger `p2` wird als

const-Zeiger deklariert und initialisiert, damit eine weitere Zuweisung an p2 durch den Compiler verhindert wird. Eine weitere Zuweisung ohne vorheriges delete [] hat den Verlust von Speicherplatz zur Folge, wie das Beispiel zeigt:

```
int *pTest   = new int[7];
pTest        = new int [10]; // 7 Elemente nicht mehr zugreifbar!
int *const pC = new int[3];
pC[1]        = 123;          // ok, die Elemente von pC sind nicht konstant
pC           = new int [33]; // Fehlermeldung des Compilers:
                             // konstantes Objekt pC kann nicht geändert werden!
```

Die 7 Elemente können nicht mehr per delete [] freigegeben werden, weil die Information über ihren Ort verloren gegangen ist. Im zweiten Fall ist zu unterscheiden, dass pC zwar konstant ist, nicht aber der Speicherbereich, auf den pC zeigt. Ein konstanter Zeiger auf ein Objekt ist etwas anderes als ein Zeiger auf ein konstantes Objekt. In Analogie zu zweidimensionalen Matrizen wird konsequenterweise eine dreidimensionale Matrix als Array von zweidimensionalen Matrizen formuliert:

```
int (*const p3)[5][7]= new int [44][5][7];
```

p3 ist ein Zeiger, der auf ein Array mit 5 Elementen zeigt, die wiederum aus einem int-Array mit 7 Elementen bestehen.[2] p3 zeigt auf das erste von 44 5x7-Teil-Arrays. Auch hier geht »[5][7]« in den Datentyp ein und muss daher links und rechts übereinstimmen. Allgemein dürfen anstelle der 5 und der 7 nur konstante Ausdrücke stehen, anstelle der 44 kann ein beliebiger ganzzahliger Ausdruck stehen.

Das Schema setzt sich für vier-, fünf-, ...-dimensionale Arrays fort – aber wer braucht die schon! Auch die Größe eines mit new erzeugten Arrays kann *nicht* mit sizeof() ermittelt werden! sizeof(p3) ergibt nur den Platzbedarf für den Zeiger selbst (zum Beispiel 4 Byte), nicht den mit new angelegten Platz.

Mehrdimensionale Arrays mit variabler Feldgröße

Mit variabler Feldgröße ist hier gemeint, dass die Größe des Arrays zur Compilierzeit nicht bekannt ist. Die Größe kann dann natürlich nicht als Teil des Datentyps aufgefasst werden. Hier wird nur kurz auf zweidimensionale Matrizen eingegangen, weil erstens drei- und mehrdimensionale Matrizen leicht daraus ableitbar sind und weil zweitens eine komfortablere Lösung in Abschnitt 9.8 vorgestellt wird. Eine mögliche Lösung besteht darin, zunächst ein Feld von Zeigern auf eindimensionale Arrays (Zeilen) anzulegen und jedem dieser Zeiger eine Zeile mit Spalten zuzuordnen (siehe Abbildung 5.12).

```
int z,s;                  // Zeilen , Spalten
cout << "Zeilen und Spalten eingeben:";
cin >> z >> s;            // erst zur Laufzeit bekannt
// Feld von Zeigern auf Zeilen anlegen:
int **const mat = new int* [z]; // mat ist ein konstanter Zeiger auf Zeiger auf int
// jeder Zeile Speicherplatz zuordnen:
for(size_t i = 0; i < z; ++i) {
   mat[i] = new int [s];
}
```

[2] Eine Anleitung zum Lesen solcher Deklarationen ist auf Seite 226 zu finden.

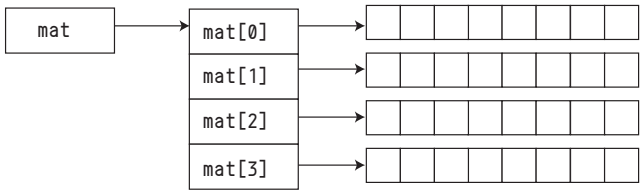

Abbildung 5.12: Zweidimensionales dynamisches Array

`mat` kann nun wie eine gewöhnliche Matrix benutzt werden. Der Zugriff auf ein Element der Matrix `mat` in Zeile `i` und Spalte `j` wird vom Compiler in die entsprechende Zeigerdarstellung umgewandelt, wie schon auf Seite 212 beschrieben.

```
// Beispiel für die Benutzung der dynamisch erzeugten Matrix
for(size_t iz = 0; iz < z; ++iz) {
    for(size_t is = 0; is < s; ++is) {
        mat[iz][is] = iz*s + is;
        cout << mat[iz][is] << '\t';
    }
    cout << endl;
}
```

Der für `mat` mit `new` angelegte Speicherbereich muss nach Gebrauch explizit wieder freigegeben werden, falls er nicht bis zum Programmende bestehen bleiben soll. Wir dürfen jedoch `delete` nicht unmittelbar auf `mat` anwenden, weil dann die einzelnen Zeilen nicht mehr freigegeben werden könnten. Damit ergibt sich folgender Ablauf:

```
// Matrix freigeben
for(size_t zeile = 0; zeile < z; ++zeile) {
    delete [] mat[zeile]; // zuerst Zeilen freigeben
}
delete [] mat;  // Feld mit Zeigern auf Zeilenanfänge freigeben
```

5.7.3 Klasse für dynamisches zweidimensionales Array

Dynamische zweidimensionale Arrays sind vielseitig einsetzbar. Aber jedes Mal Speicher zuzuweisen und für ein korrektes `delete` zu sorgen, ist umständlich. Es bietet sich an, diese Vorgänge in einer Klasse zu kapseln mit dem Ziel, dass Objekte dieser Klasse ohne Performance-Verlust benutzt werden können, so wie hier gezeigt:

Listing 5.6: Anwendung für Array2d

```
// cppbuch/k5/array2d/main.cpp
#include"array2d.h"          // Klasse Array2d
#include<iostream>
using namespace std;

int main() {
   Array2d<int> arr(5, 7);
   arr.init(0);
   printArray(arr);                    // Ausgabe mit Hilfsfunktion
   for(size_t z = 0; z < arr.getZeilen(); ++z) {
      for(size_t s = 0; s < arr.getSpalten(); ++s) {
```

```
            arr.at(z, s) = 10*z+s;      // Benutzung, schreibend und ...
            cout << arr.at(z, s) << " "; // lesend (siehe Text unten)
        }
        cout << endl;
    }
    Array2d<int> arr1(arr); // Kopierkonstruktor
    printArray(arr1);
    arr1.init(3);           // Neuinitialisierung
    arr.assign(arr1);       // Zuweisung
    printArray(arr);
    Array2d<double> arrd(3, 4, 99.013); // double-Array
    printArray(arrd);
    Array2d<string> arrs(2, 5, "hello"); // string-Array
    printArray(arrs);
}
```

Wie man sieht, dürfen die Array-Elemente nicht nur `int`- oder `double`-Zahlen sein, sondern auch Strings. In der Datei *cppbuch/k5/array2d/array2d.h* übernimmt der Konstruktor die Speicherallokation, der Destruktor sorgt für die Freigabe. Weil das Überladen von Operatoren erst im Kapitel 9 behandelt wird, gibt es hier noch einige Einschränkungen:

- Anstelle der Schreibweise `arr = arr1;` für die Zuweisung wird `arr.assign(arr1);` geschrieben.
- Der Zugriff auf ein Array-Element über den Indexoperator wird durch einen Funktionsaufruf ersetzt, also etwa `arr.at(z, s) = wert;` statt `arr[z][s] = wert;`.

Entwurfsüberlegungen

Da der schreibende Funktionsaufruf auf der *linken* Seite der Zuweisung steht, ergibt sich, dass er eine Referenz auf das zu ändernde Array-Element zurückgeben muss. Bei dem lesenden Funktionsaufruf, der das Array-Objekt `arr` nicht ändert, genügt eine Kopie des Werts oder eine `const`-Referenz.

Falls das Array innerhalb der Klasse so wie in Abbildung 5.12 (Seite 215) erzeugt wird, könnte `at(i,j)` wie folgt realisiert werden, wobei `T` der Typ der Array-Elemente und `ptr` der Zeiger auf den Beginn des internen Arrays ist:

```
T& at(size_t z, size_t s) {
    return ptr[z][s]; // ptr ist hier vom Typ T**.
}
```

Konstruktor und Destruktor wären wegen der Schleifen etwas umständlich. Aus diesem Grund kann die Beschaffung des Speichers als *ein* Block erwogen werden, zum Beispiel `T* ptr = new T[zeilen*spalten];`. `ptr` wäre dann vom Typ `T*` und nicht mehr `T**`. Im Destruktor genügte ein einziges `delete []`. Das bedeutet allerdings, dass ein Zugriff wie `ptr[z][s]` nicht möglich ist. Die Adresse müsste anders berechnet werden – wie, das zeigt die Lösung der Aufgabe 5.2 von Seite 213. Damit wird die Funktion `at()` wie folgt realisiert (`spalten` ist die Anzahl der Spalten):

```
T& at(size_t z, size_t s) {
    return ptr[z * spalten + s]; // ptr ist hier vom Typ T*.
}
```

Der Compiler würde im ersten Fall `ptr[z][s]` in `*(*(ptr+z)+s)` umwandeln, wie auf Seite 212 erläutert. Weil Konstruktor und Destruktor einfacher werden und die Berechnung der Adresse etwa ebenso schnell ist wie die Berechnung von `*(*(ptr+z)+s)`, wird der zweiten Variante, also der Beschaffung des Speichers in einem einzigen Schritt, der Vorzug gegeben. Das folgende Listing zeigt die dazu passende Klasse:

Listing 5.7: Klasse Array2d

```
// cppbuch/k5/array2d/array2d.h
#ifndef ARRAY2D_H
#define ARRAY2D_H
#include<cassert>
#include<iostream>

template<typename T>
class Array2d {
public:
  Array2d(size_t z, size_t s)
    : zeilen(z), spalten(s), ptr(new T[z*s]) {
  }

  Array2d(size_t z, size_t s, const T& wert)
    : zeilen(z), spalten(s), ptr(new T[z*s]) {
    init(wert);
  }

  Array2d(const Array2d& a)
    : zeilen(a.zeilen), spalten(a.spalten),
      ptr(new T[zeilen*spalten]) {
    size_t anzahl = zeilen * spalten;
    for(size_t i=0; i < anzahl; ++i) {
      ptr[i] = a.ptr[i];
    }
  }

  ~Array2d() {
    delete [] ptr;
  }

  Array2d& assign(Array2d tmp) {  // siehe Text unten
    swap(tmp);
    return *this;
  }

  size_t getZeilen() const { return zeilen; }
  size_t getSpalten() const { return spalten; }

  void init(const T& wert) {  // Alle Elemente mit wert initialisieren
    size_t anzahl = zeilen * spalten;
    for(size_t i=0; i < anzahl; ++i) {
      ptr[i] = wert;
    }
  }
```

```
   const T& at(size_t z, size_t s) const { // für lesenden Zugriff
      assert(z < zeilen && s < spalten);
      return ptr[z * spalten + s];
   }

   T& at(size_t z, size_t s) {            // für verändernden Zugriff
      assert(z < zeilen && s < spalten);
      return ptr[z * spalten + s];
   }

   void swap(Array2d& rhs) { // Daten von *this mit denen von rhs vertauschen, s.u.
      size_t temp = zeilen;
      zeilen = rhs.zeilen;
      rhs.zeilen = temp;
      temp = spalten;
      spalten = rhs.spalten;
      rhs.spalten = temp;
      T* tempPtr = ptr;
      ptr = rhs.ptr;
      rhs.ptr = tempPtr;
   }
private:
   size_t zeilen;
   size_t spalten;
   T* ptr;
   Array2d& operator=(const Array2d& arr); // (noch) verbieten
};

// Globale Funktion zur Ausgabe
template<typename T>
void printArray(const Array2d<T>& a) {
   for(size_t z=0; z < a.getZeilen(); ++z) {
      for(size_t s=0; s < a.getSpalten(); ++s) {
         std::cout << a.at(z,s) << " ";
      }
      std::cout << std::endl;
   }
}
#endif
```

Die private Deklaration des Zuweisungsoperators verhindert, dass `arr = arr1;` geschrieben werden kann. Wie er realisiert werden kann, erfahren Sie im Kapitel 9. In der Funktion `assign(arr)` müsste der Ablauf die folgenden Schritte umfassen: 1. Speicher beschaffen, 2. Werte kopieren, 3. alten Speicher freigeben, 4. Verwaltungsdaten aktualisieren.

Zu den ersten beiden Schritten findet sich Code im Kopierkonstruktor, zum dritten Schritt im Destruktor. Um eine Codeduplikation zu vermeiden, wird in `assign()` eine lokale Kopie durch die Übergabe des Parameters per Wert angelegt. Durch die lokale Kopie statt einer Referenz wird erreicht, dass sie ohne Auswirkung auf das Original verändert werden kann Die Funktion `swap()` sorgt für die Vertauschung der Objektinhalte (es gibt auch eine `swap()`-Bibliotheksfunktion). Danach hat `*this` den Inhalt von `arr` (= Sinn der Zuweisung) und `arr` den von `*this`. Letzteres wird zurückgegeben, und der Destruktor von

`tmp` sorgt für die Speicherfreigabe. Die Abbildungen 5.13 bis 5.15 zeigen die Schritte für eine hypothetische Anweisung `x.assign(y);`. Die Ellipsen stehen für die `Array2d`-Objekte. Das Objekt `x` kann innerhalb der Funktion mit `*this` angesprochen werden. Nach dieser Anweisung gilt `x == y`.

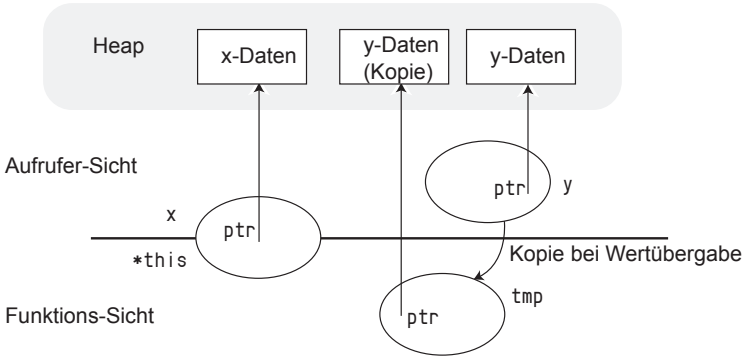

Abbildung 5.13: Zustand direkt nach Funktionsaufruf, aber vor `swap()`

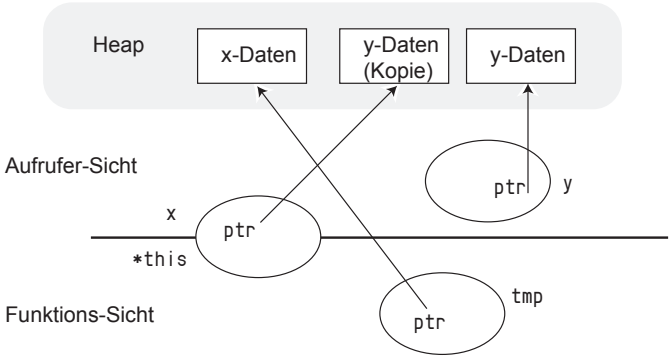

Abbildung 5.14: Zustand nach `swap()`

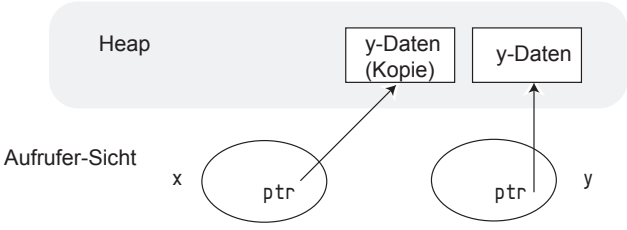

Abbildung 5.15: Zustand nach Funktionsende. Der Destruktor hat `tmp` zerstört.

5.8 Binäre Ein-/Ausgabe

Im Gegensatz zur formatierten Ein-/Ausgabe mit den Operatoren << und >> ist der binäre Datentransfer *unformatiert*. Unformatiert heißt, dass die Daten in der *internen* Darstellung direkt geschrieben beziehungsweise gelesen werden, eine Umwandlung zum Beispiel einer `float`-Zahl in eine Folge von ASCII-Ziffernzeichen also unterbleibt. Auf diese Art beschriebene Dateien können nicht sinnvoll mit einem Texteditor bearbeitet oder direkt ausgedruckt werden. Die zum binären Datentransfer geeigneten Funktionen in C++ sind `read()` und `write()`.

Das Prinzip: Es wird ein Zeiger auf den Beginn des Datenbereichs angegeben, also die Adresse des Bereichs. Dabei wird der Zeiger in den Datentyp `char*` umgewandelt. `write()` verlangt an dieser Stelle `char*`, weil ein `char` einem Byte entspricht. Zusätzlich wird die Anzahl der zu transferierenden Bytes angegeben. Zur Wandlung des Zeigers wird der `reinterpret_cast`-Operator verwendet. Dieser Operator verzichtet im Gegensatz zum `static_cast`-Operator von Seite 53 auf jegliche Verträglichkeitsprüfung, weil hier Zeiger auf *beliebige*, das heißt auch selbst geschriebene Datentypen in den Typ `char*` umgewandelt werden sollen.

Das Beispiel zeigt das unformatierte Schreiben und Lesen von `double`-Zahlen als Vorlage, wie man es machen kann. Der Dateiname wurde nur deswegen fest vorgegeben, um die Programme nicht so lang werden zu lassen. Das Schema lässt sich sinngemäß auf beliebige Datentypen und Mengen übertragen. Die `for`-Schleife im ersten Beispielprogramm zeigt eine typische Anwendung des auf den Seiten 76 und 210 erwähnten Kommaoperators. Die zu schreibenden Zahlen werden nur als Beispiel nach der Vorschrift 1.1^i, $i = 0..19$ berechnet.

Listing 5.8: Schreiben einer binären Datei

```
// cppbuch/k5/wdouble.cpp  Erzeugen einer Datei double.dat mit 20 double-Zahlen
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>
using namespace std;

int main( ) {
   ofstream ziel;
   // ios::binary mit ios::out verwenden (siehe S. 390).
   ziel.open("double.dat", ios::binary|ios::out);
   if(!ziel) {
      cerr << "Datei kann nicht geöffnet werden!\n";
      exit(-1);
   }
   double d = 1.0;
   for(int i = 0;i < 20; ++i, d *= 1.1) // Schreiben von 20 Zahlen
     ziel.write(reinterpret_cast<const char*>(&d), sizeof(d));
}  // ziel.close() wird vom Destruktor durchgeführt
```

Listing 5.9: Lesen einer binären Datei

```
// cppbuch/k5/rdouble.cpp
```

```
// Lesen einer Datei double.dat mit double-Zahlen
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>
using namespace std;
int main( ) {
    ifstream quelle;
    quelle.open("double.dat", ios::binary|ios::in);
    if(!quelle) { // muss existieren
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    double d;
    while(quelle.read(reinterpret_cast<char*>(&d), sizeof(d)))
        cout << " " << d << '\n';
}  // quelle.close() wird vom Destruktor durchgeführt
```

Im zweiten Beispiel wird beim Lesen die Variable `d` verändert. Deswegen fehlt im Vergleich zum ersten Programm das `const` in der Typumwandlung.

ASCII oder binär?

Um den Unterschied zu verdeutlichen, folgt ein Beispiel, in dem eine Matrix einmal als ASCII-Datei und einmal als binäre Datei ausgegeben wird. Die Merkmale der Dateitypen sind im Wesentlichen:

ASCII-Datei

- ASCII-Dateien sind mit einem Texteditor lesbar.
- Schreiben und Lesen von ASCII-Dateien mit einem Programm dauern im Allgemeinen länger als binäres Schreiben oder Lesen von binären Dateien, weil implizit Umformatierungen vom internen Format nach ASCII (oder umgekehrt) vorgenommen werden.
- Anstelle der Funktionen `get()` und `put()` für zeichenweise Verarbeitung können auch die Operatoren >> und << zur Ein- und Ausgabe verwendet werden. In der Anwendung besteht kein Unterschied zur Standardein- und -ausgabe.
- Die Dateigröße bestimmt sich aus der Anzahl der ausgegebenen Zeichen und hängt damit von der Formatierung ab. Die Ausgabe sollte so formatiert sein, dass ein problemloses Einlesen möglich ist, zum Beispiel durch Angabe einer genügenden Ausgabeweite oder Einfügen von Leerzeichen zur Trennung.
- Durch die Formatierung kann es zu Wertänderungen kommen, zum Beispiel durch Abschneiden von Nachkommastellen.

Listing 5.10: Dateiausgabe eines Arrays

```
// cppbuch/k5/wmatrix.cpp
// Schreiben einer Matrix als ASCII- und als binäre Datei
// (siehe auch Übungsaufgabe)
#include<cstdlib>
#include<fstream>
#include<iostream>
```

```
using namespace std;

int main() {
    const int ZEILEN = 10, SPALTEN = 8;
    double matrix[ZEILEN][SPALTEN];
    for(int i = 0; i < ZEILEN; ++i)// Matrix mit Werten füllen
        for(int j = 0; j < SPALTEN; ++j)
            matrix[i][j] = i+1 + (j+1)/1000.0;

    // Schreiben als ASCII-Datei (lesbar mit Editor)
    ofstream ziel;
    ziel.open("matrix.asc");
    if(!ziel) {
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    // formatiertes Schreiben
    for(int i = 0; i < ZEILEN; ++i) {
        for(int j = 0; j < SPALTEN; ++j) {
            ziel.width(8);     // siehe auch Seite 378
            ziel << matrix[i][j];
        }
        ziel << endl;
    }
    // Datei schließen, damit ziel wieder verwendet werden kann:
    ziel.close();

    // Schreiben als binäre Datei
    ziel.open("matrix.bin", ios::binary|ios::out);
    if(!ziel) {
        cerr << "Datei kann nicht geöffnet werden!\n";
        exit(-1);
    }
    ziel.write(reinterpret_cast<const char*>(matrix),
               sizeof(matrix));
}   // automatisches close()
```

Binärdatei

- Binärdateien sind *nicht* mit einem Texteditor lesbar und auch nicht druckbar.
- Schreiben und Lesen von Binärdateien ist schnell, weil Umformatierungen nicht notwendig sind. Voraussetzung des Vergleichs ist natürlich, dass die Ein- und Ausgabe beider Arten von Dateien auf dieselbe Weise vom Betriebssystem gepuffert wird.
- Die Dateigröße bestimmt sich aus der Größe und Anzahl der ausgegebenen Daten und hat nichts mit ihrer ASCII-Darstellung zu tun, die je nach Formatierung auf verschiedene Art möglich ist. Im Programm oben ist die Dateigröße in Bytes gleich der Matrixgröße `sizeof(double)`·(Anzahl der Zeilen)·(Anzahl der Spalten).
- Große Dateistrukturen können ohne Benutzung von Schleifen mit einer *einzigen* Anweisung ausgegeben werden (siehe `matrix` im Beispiel).

5.9 Zeiger auf Funktionen

Zeiger auf Funktionen ermöglichen es, erst zur Laufzeit zu bestimmen, welche Funktion ausgeführt werden soll (englisch *late binding, dynamic binding*) – die Grundlage für den noch zu besprechenden Polymorphismus. Im Beispielprogramm wird entweder das Maximum oder das Minimum der Zahlen 1700 und 1000 ausgegeben. Welche Funktion tatsächlich gewählt wird, entscheidet sich erst *zur Laufzeit*, sodass zur Compilationszeit nicht bekannt sein *kann*, ob mit `(*fp)(a,b)` im Beispielprogramm `max()` oder `min()` aufgerufen wird.

Zur Deklaration: `fp` ist ein Zeiger auf eine Funktion, die einen `int`-Wert zurückgibt und zwei `int`-Parameter verlangt. Dabei ist `(*fp)` statt `*fp` notwendig, um eine Verwechslung mit einer Funktionsdeklaration `int *fp (int, int)` zu vermeiden.

Zeiger auf Funktionen können wie andere Variablen anderen Funktionen als Parameter übergeben werden. Zum Beispiel könnte man einer Funktion, die mathematische Graphen zeichnet, wahlweise einen Zeiger auf die Sinus-Funktion oder auf eine Hyperbel-Funktion übergeben. Das folgende Beispiel zeigt, wie wir eine vordefinierte Funktion mithilfe von Zeigern auf Funktionen nutzen.

Listing 5.11: Funktionszeiger

```
// cppbuch/k5/funkptr.cpp
#include<cstddef>
#include<iostream>
using namespace std;

int max(int x, int y) { return x > y ? x : y;}
int min(int x, int y) { return x < y ? x : y;}

int main() {
  int a = 1700, b = 1000;
  int (*fp)(int, int); // fp ist Zeiger auf eine Funktion
  do {
    char c;
    cout << "max (1) oder min (0) ausgeben (sonst = Ende)?";
    cin >> c;
    // Zuweisung von max() oder min()
    switch(c) {
      case '0': fp = &min; break; // Funktionsadresse zuweisen
      // Ohne den Adressoperator & wandelt der Compiler den
      // Funktionsnamen automatisch in die Adresse um:
      case '1': fp = max; break;
      default : fp = NULL;
    }
    if(fp) {              // d.h if(fp != 0)
      // Dereferenzierung des Funktionszeigers und Aufruf
      cout << (*fp)(a, b) << endl;
      // oder direkt Funktionszeiger als Name verwenden:
      // (implizite Typumwandlung des Zeigers in die Funktion)
      cout << fp(a, b) << endl;
```

```
    }
  } while(fp);
}
```

Wenn wir ein C-Array sortieren wollen, müssen wir keine eigene Funktion schreiben, sondern können die Funktion `qsort()` verwenden, die in jedem C++-System vorhanden und deren Prototyp im Header `<cstdlib>` deklariert ist. Die hier vorgestellte Bibliotheksfunktion `qsort()` unterscheidet sich von der Funktion `quicksort()` von Seite 134 dadurch, dass die Allgemeinheit der Funktion über ihre Realisierung mit Zeigern auf `void` hergestellt wird und nicht über ein Template. Außerdem sortiert sie C-Arrays und keine `vector`-Objekte.

```
void qsort(void *feldname,
           size_t zahlDerElemente,
           size_t bytesProElement,
           int (*cmp)(const void *a, const void *b));
```

Hinweis

Hier geht es um die Wirkungsweise von Funktionszeigern, nicht um das Sortieren. Wenn es nur um Sortieren geht, wird am besten die Funktion `std::sort()` benutzt, die an anderer Stelle beschrieben wird (Abschnitt 24.4.2).

Der letzte Parameter ist ein Zeiger auf eine Funktion, die einen `int`-Wert zurückgibt und zwei Zeiger auf `const void` als Parameter verlangt, hier `a` und `b` genannt. Der Name `cmp` steht für »compare« und ist hier nur zu Dokumentationszwecken vorhanden; er kann auch weggelassen werden.

`qsort()` soll als Standardfunktion für beliebige Arrays geeignet sein, daher der Datentyp `void*`. Die Funktion mit dem Platzhalternamen `cmp` ist von uns zu schreiben. Sie hat die Aufgabe zu entscheiden, welches von den beiden Arrayelementen, auf die die Zeiger `a` und `b` zeigen, größer ist. Als Ergebnis für eine aufsteigende Sortierung wird verlangt, dass ein Wert kleiner 0 zurückgegeben wird, falls `*a < *b` ist, ein Wert größer 0, falls `*a > *b` ist, und 0 bei `*a == *b`.

Bei absteigender Sortierung sind die Vorzeichen zu vertauschen. Die Funktion zum Vergleich und ihr Aufruf sind unten am Beispiel des Sortierens eines `int`-Feldes beschrieben. Die Schnittstelle muss zu `qsort()` passen, deshalb wird in der Funktion eine Typumwandlung vorgenommen. Dies geschieht dadurch, dass der Zeiger auf `void` in einen Zeiger auf `int` umgewandelt wird, über den durch Dereferenzierung (`*`) die zu vergleichenden Zahlenwerte `ia` und `ib` gewonnen werden. Ein Umweg der Art

```
int (*vergleich)(const void*, const void*);
vergleich = icmp;
qsort(ifeld, anzahlDerElemente, sizeof(ifeld[0]), vergleich);
```

ist nicht notwendig; der Name der Funktion `icmp` kann direkt eingesetzt werden. Die Bedeutung der Parameterliste im Aufruf ergibt sich durch Vergleich mit dem oben angegebenen Prototypen. Den Quellcode von `qsort()` kennen wir nicht und brauchen ihn auch nicht zu kennen, um `qsort()` benutzen zu können. Es genügt ausschließlich die Kenntnis der Schnittstelle und natürlich die Kenntnis dessen, was `qsort()` tut. Wie in

qsort() die Funktion icmp angesprochen wird, ist letztlich nicht wichtig zu wissen und wird hier nur zur Erläuterung beschrieben. Innerhalb von qsort() steht an irgendeiner Stelle etwa sinngemäß (wobei die Namen im uns unbekannten tatsächlichen Quellcode natürlich ganz anders lauten können):

```
if(icmp(feld+i, feld+j) < 0) {
  ...
}
else {
   // ...usw.
}
```

Durch die Übergabe des Funktionszeigers »weiß« qsort(), welche Funktion zu nehmen ist, ganz in Analogie zum oben aufgeführten min-max-Beispiel. Dieser Mechanismus wird *callback* genannt und immer dann eingesetzt, wenn ein Server eine Dienstleistung erbringen soll und dazu eine Funktion oder Methode des Client benötigt, die er nicht automatisch zur Verfügung hat. Der Server ist in diesem Fall die Funktion qsort(), die die Dienstleistung (= das Sortieren) nur erbringen kann, wenn der Client mitteilt, wie die Objekte zu vergleichen sind. Der Server kann dann die ihm (hier über einen Funktionszeiger) mitgeteilte Funktion aufrufen (= callback), ohne ihre Details zu kennen.

Listing 5.12: Quicksort und Funktionszeiger

```
// cppbuch/k5/qsort.cpp
#include<iostream>
#include<cstdlib> // enthält Prototyp von qsort()
using namespace std;

// Definition der Vergleichsfunktion
int icmp(const void *a, const void *b) {
    // Typumwandlung der Zeiger auf void in Zeiger auf int
    // und anschließende Dereferenzierung (von rechts lesen).
    // Die Typumwandlung meint, dass der Speicherinhalt,
    // auf den a und b verweisen, als int zu interpretieren ist.
    int ia = *static_cast<const int*>(a);
    int ib = *static_cast<const int*>(b);
    // Vergleich und Ergebnisrückgabe ( > 0, = 0, oder < 0)
    if(ia == ib)
       return 0;
    return ia > ib? 1 : -1;
}

int main() {
    int ifeld[] = {100, 22, 3, 44, 6, 9, 2, 1, 8, 9};
    // Anzahl der Elemente = sizeof(Feld) / sizeof(ein Element)
    size_t anzahlElemente = sizeof(ifeld)/sizeof(ifeld[0]);
    // Aufruf von qsort() mit Funktionszeiger icmp:
    qsort(ifeld, anzahlElemente, sizeof(ifeld[0]), icmp);
    for(size_t i = 0; i < anzahlElemente; ++i) // Ausgabe des sortierten Feldes
        cout << ' ' << ifeld[i];
    cout << endl;
}
```

Deklarationen mit Zeigern auf Funktionen sind manchmal etwas mühsam zu lesen. Hinweise zum Lesen komplexer Deklarationen werden auf Seite 226 gegeben.

Übungen

5.5 Schreiben Sie eine Funktion `strcopy(char* ziel, const char* quelle)`, die den Rückgabetyp `void` hat und die den Inhalt des Strings `quelle` in den String `ziel` kopiert, wobei der vorherige Inhalt von `ziel` dabei überschrieben wird. Es sei vorausgesetzt, dass `ziel` ausreichend groß ist, um `quelle` aufzunehmen.

5.6 Schreiben Sie eine Funktion `char* strduplikat(const char* s)`, die den String `s` dupliziert, indem neuer Speicherplatz mit `new` beschafft und `s` in diesen Bereich hineinkopiert wird. Ein Zeiger auf den Beginn des Duplikats soll zurückgegeben werden.

5.7 Auf Seite 224 wurde gezeigt, wie Quicksort auf ein Ganzzahlen-Feld angewendet werden kann. Gegeben sei nun ein alphabetisch zu sortierendes String-Array.

```
const char *sfeld[]={"eins", "zwei", "drei", "vier", "fünf",
                     "sechs", "sieben", "acht", "neun", "zehn"};
```

Schreiben Sie ein Programm, das dieses Array mit `qsort()` sortiert. Wie sieht die benötigte Vergleichsfunktion aus, die hier `scmp()` genannt sei? Wie ist `qsort()` aufzurufen?

Hinweise: a) Diese Aufgabe ist nur für diejenigen, die ihr Wissen über den Einsatz von Funktionszeigern bei einer C-Bibliotheksfunktion vertiefen möchten – C ist schließlich eine Untermenge von C++. Die anderen benutzen besser `std::sort()`. Ein Beispiel für die Alternative finden Sie nach der Lösung zu dieser Aufgabe (siehe Seite 914). b) Innerhalb von `scmp()` kann `strcmp()` aus `<cstring>` benutzt werden. `strcmp(a,b)` gibt eine Zahl kleiner 0 zurück, falls der C-String `a` < der C-String `b` ist, größer 0, falls `a` > `b` ist, und 0 bei Gleichheit.

5.10 Komplexe Deklarationen lesen

Komplexe Deklarationen sind manchmal schwer zu lesen. Es gibt dafür ein Rezept, das hier etwas vereinfacht dargestellt wird: Man löst die Deklaration vom Namen her auf und schaut dann nach rechts und links in Abhängigkeit vom Vorrang der Syntaxelemente. Beginnend bei den wichtigen Elementen ist die Reihenfolge (nach [vdL]):

- runde Klammern ( ), die Teile der Deklaration zusammenfassen (gruppieren);
- runde Klammern ( ), die eine Funktion anzeigen;
- eckige Klammern [ ], die ein Array anzeigen;
- Sternsymbol *, das »Zeiger auf« bedeutet.

Beispiel:

```
char *(*(*seltsam)(double, int))[3]
```

Name: `seltsam`

Klammer: Die Klammer dient zur Gruppierung, das nächste Zeichen ist daher *.

`*`: `seltsam` ist ein Zeiger. Nun haben wir die Auswahl zwischen einer Funktion (Klammer rechts) oder einem Zeiger (Sternchen links). Die Klammer hat Vorrang.

`(double, int)`: `seltsam` ist ein Zeiger auf eine Funktion, die über ein `double`- und ein `int`-Argument verfügt.

Klammer: Die Klammer dient zur Gruppierung, das nächste Zeichen ist `*`. Die Funktion gibt also einen Zeiger zurück.

eckige Klammer: Vorrang gegenüber `*`. Der Zeiger zeigt auf ein Array mit 3 Elementen,

`*`, links: die `char`-Zeiger sind.

Zusammengefasst: `seltsam` ist ein Zeiger auf eine Funktion mit einem `double`- und einem `int`-Argument, die einen Zeiger auf ein Array von Zeigern auf `char` zurückgibt. Ein weiteres Beispiel: `int** was[2][3];`:
`was` ist ein Array von Arrays (= zweidimensionales Array), dessen Elemente Zeiger auf Zeiger auf `int` sind. Eine gut lesbare und ausführliche Behandlung des Themas ist in [vdL] zu finden.

typedef

Komplizierte Deklarationen sollten der schlechten Lesbarkeit wegen vermieden werden. Manchmal sind sie jedoch unumgänglich. Für diesen Fall bietet das Schlüsselwort `typedef` die Möglichkeit, die Lesbarkeit durch Strukturierung der Namensgebung zu verbessern. Betrachten wir zunächst ein einfaches Beispiel: In einem Programm wollen wir die Unterschiede in der Genauigkeit der Ergebnisse je nach Datentyp `float` oder `double` untersuchen. Mit `typedef` können wir uns einen anderen Namen für den Datentyp schaffen:

```
typedef float real;
int main() {
   real Zahl = 1.7353;  // Zahl ist vom Typ float
   // ...
}
```

Wenn wir dasselbe Programm mit `double`-Zahlen rechnen lassen wollen, brauchen wir nur die `typedef`-Zeile durch

```
typedef double real;
```

zu ersetzen und neu zu compilieren. Alle Vorkommen von `real` bleiben unverändert, haben aber nun die Bedeutung von `double`.

Komplexe Deklarationen lassen sich durch `typedef` lesbarer gestalten, weil neue Typnamen zur Strukturierung benutzt werden können. Das obige `seltsame` Beispiel sei wieder aufgegriffen, indem ein neuer Datentyp `ArrayVon3CharZeigern` definiert wird:

```
typedef char* ArrayVon3CharZeigern [3];
// die Deklaration
ArrayVon3CharZeigern A;  // ist identisch mit:
char* A[3];
```

Der Rückgabetyp der Funktion, die zum Funktionszeiger `seltsam` der Vorseite passt, ist ein Zeiger auf den neuen Datentyp:

```
typedef ArrayVon3CharZeigern *ZeigerAufArrayVon3CharZeigern;
```

Äquivalent dazu ist:

```
typedef char* (*ZeigerAufArrayVon3CharZeigern) [3];
```

Der Datentyp des Zeigers `seltsam` kann mit `typedef` beschrieben werden:

```
typedef char*(*(*seltsamerTyp)(double,int))[3];
```

oder erheblich lesbarer mit

```
typedef ZeigerAufArrayVon3CharZeigern (*seltsamerTyp)(double,int);
```

Damit ließe sich eine »seltsame Funktion« deklarieren, die über die Zeiger ausgeführt werden kann:

```
// Prototyp (Implementation ist sonstwo...)
ZeigerAufArrayVon3CharZeigern seltsameFunktion(double,int);
ZeigerAufArrayVon3CharZeigern z;
seltsam = seltsameFunktion;
z  = seltsam(3.1, 2);  // Ausführung über Zeiger
seltsamerTyp X;
X = seltsam;
z  = X(3.1, 2);        // Ausführung über Zeiger
```

In [vdL] ist ein C-Programm abgedruckt, das in der Unix-Welt unter dem Namen *cdecl* bekannt ist. Es übersetzt komplizierte Deklarationen in verständlichen Text. Das Programm ist auch in den Beispielen der DVD vorhanden, siehe *cppbuch/k5/cdecl.cpp*. Das Programm wartet nach dem Start auf die Eingabe einer Deklaration, zum Beispiel `char* A[3]`, und gibt danach die Auswertung aus (»A is array 0..2 of pointer to char«).

Übungen

5.8 Schreiben Sie eine Funktion `void leerzeichenEntfernen(char* s)`, die alle Leerzeichen im `char`-Array `s` entfernt. Zum Beispiel soll aus »a bb ccc d« die Zeichenkette »abbcccd« werden. Verwenden Sie dabei Zeiger.

5.9 Schreiben Sie ein Programm, das Dateien, deren Namen in der Kommandozeile angegeben werden, auf der Standardausgabe ausgibt. Zum Beispiel könnte der Befehl

```
prog datei1.cpp XXX datei2.cpp
```

dazu führen, dass die Dateien *datei1.cpp* und *datei2.cpp* auf dem Bildschirm angezeigt werden. Wenn eine Datei XXXX nicht vorhanden ist, soll es eine Fehlermeldung »Datei XXXX nicht gefunden!« geben.

5.10 Schreiben Sie ein Programm, das alle in einer Datei vorkommenden Namen ausgibt. Ein Name ist dabei so definiert: Er beginnt mit einem Buchstaben, anschließend folgen beliebig viele Buchstaben und Ziffern. Der Unterstrich zählt auch als Buchstabe.

5.11 Standard-Typumwandlungen für Zeiger

- Ein integraler Ausdruck, der 0 ergibt, kann zu einem Zeiger auf ein Objekt eines beliebigen Typs T konvertiert werden: `T* ptr = 0;` (Null-Zeiger).
- Jeder Zeiger ist nach `bool` konvertierbar. Dabei wird ein Null-Zeiger stets zu `false` und ein anderer Zeiger stets zu `true` ausgewertet.
- Jeder Zeiger auf ein Objekt eines beliebigen Typs T kann in einen Zeiger auf `void` konvertiert werden:

```
T* ptr;
void* vptr = ptr;
```

  Anmerkung: Die in Abschnitt 5.12 beschriebenen Zeiger auf Elementfunktionen und -daten sind keine Zeiger auf Objekte und können nicht nach `void*` gewandelt werden.
- Jeder Zeiger auf ein Objekt einer Klasse `B` kann in einen Zeiger auf Klasse `A` umgewandelt werden, wenn `A` eine Oberklasse von `B` ist und (bei Mehrfachvererbung) eindeutig ermittelt werden kann (die Begriffe Oberklasse und Vererbung werden in Kapitel 7 erläutert).
- Arrays können zu Zeigern konvertiert werden. Beispiel:

```
char dasArray[]="ABC";
const char* cpc = dasArray; // Array → Zeiger
char* cp = dasArray;        // Array → Zeiger
```

Funktionszeiger

Eine Funktion kann in einen Zeiger auf eine Funktion oder umgekehrt umgewandelt werden. Ein Beispiel verdeutlicht die beiden Fälle:

```
void drucke(int x) { cout << x << endl;} // Funktion

int main() {
  void (*f1)(int) = &drucke; // keine Umwandlung
  void (*f2)(int) = drucke; // Funktion → Zeiger
  f1(3);                    // Zeiger → Funktion
  (*f1)(3);                 // explizit: Zeiger→ Funktion
}
```

Hinweis: Bei Zeigern auf Elementfunktionen ist keine implizite Typumwandlung vorgesehen. In der Zeile

```
int (Ort::*fp)() const = &Ort::X;
```

des Beispielprogramms im nächsten Abschnitts darf also der Adressoperator `&` nicht weggelassen werden.

5.12 Zeiger auf Elementfunktionen und -daten[3]

C++ erlaubt es, Zeiger auf Elementdaten (englisch *data members*) und -funktionen (englisch *member functions*) zu richten. Deklaration und Zugriff auf Elemente werden mit den Operatoren `::*`, `.*` und `->*` bewerkstelligt. Diese Zeiger unterliegen einer Typprüfung durch den Compiler, die die Klassenzugehörigkeit berücksichtigt, im Unterschied zu den Zeigern auf globale Funktionen des Abschnitts 5.9.

Eine sehr große Flexibilität wird erreicht, weil Zeiger zur Laufzeit auf verschiedene Elementfunktionen (bzw. seltener Attribute) gerichtet werden können. Diese Zeiger können wiederum mit Zeigern kombiniert werden, die zur Laufzeit auf verschiedene Objekte verweisen. Die verschiedenen Objekte können sogar unterschiedlichen Typs sein, wie noch gezeigt wird (Abschnitt 7.6).

5.12.1 Zeiger auf Elementfunktionen

Die von Seite 157 bekannte Klasse `Ort` hat einige Elementfunktionen, auf die im folgenden Beispielprogramm über Zeiger zugegriffen wird. Bei der Initialisierung von Zeigern auf Elementfunktionen und -daten darf im Gegensatz zu Zeigern auf andere Funktionen (siehe Beispiel auf Seite 223) der Adressoperator `&` nicht weggelassen werden.

Listing 5.13: Zeiger auf Elementfunktionen

```
// cppbuch/k5/elfunptr.cpp
#include"ort.h"      // schließt <iostream> ein
using namespace std;

int main() {
    Ort einOrt(100,200);
    // Zeiger auf Ort::getX() richten.
    int (Ort::*fp)() const = &Ort::getX;
    cout << (einOrt.*fp)() << endl;  // dasselbe wie einOrt.getX()
    fp = &Ort::getY;                 // Funktionszeiger umschalten
    cout << (einOrt.*fp)() << endl;  // jetzt dasselbe wie einOrt.getY()
    // Das Umschalten des Zeigers auf einen neuen Ort beeinflusst nicht die
    // Zuordnung zu der Methode:
    Ort * derZeiger = new Ort(300,400);
    cout << (derZeiger->*fp)() << endl;
    // ist dasselbe wie derZeiger->getY()

    void (Ort::*elFktPtr)(int,int) = &Ort::aendern;
    (derZeiger->*elFktPtr)(500,600);
    // ist dasselbe wie derZeiger->aendern(500,600)

    anzeigen(*derZeiger);        // geänderter Ort
    delete derZeiger;
}
```

[3] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

5.12.2 Zeiger auf Elementdaten

Sicher kommt es kaum vor, dass auf Elementdaten über Zeiger zugegriffen werden soll, weil Attribute in der Regel privat sind. Das folgende Beispielprogramm zeigt, wie der Zugriff bei öffentlichen Attributen möglich ist. Man sieht dabei, dass das Umschalten auf ein neues Objekt nicht die Zuordnung zum Attribut `b` zerstört.

Listing 5.14: Zeiger auf Elementdaten

```
// cppbuch/k5/eldatptr.cpp
#include<iostream>
using namespace std;

struct Datensatz {                          // öffentliche Klasse
      Datensatz(int x, int y)               // Konstruktor
      : a(x), b(y) {}
      int a;
      int b;
};

int main() {
    // Zeiger auf Elementattribute
    Datensatz einDatensatz(1, 2);
    int Datensatz::*dp = &Datensatz::a;

    cout << einDatensatz.*dp << endl; // 1
    dp = &Datensatz::b;             // Zeiger umschalten
    cout << einDatensatz.*dp << endl; // 2

    // neues Objekt erzeugen
    Datensatz* dPtr = new Datensatz(1000, 2000);
    cout << dPtr->*dp << endl;      // 2000
    delete dPtr;
}
```