

dieses Programms. Testen Sie das Programm mit und ohne `freeze(0)`-Anweisungen.

4. Implementieren Sie Ihre Version des Linux-Werkzeuges `wc`. Schreiben Sie also ein Programm, das den Namen einer Textdatei über die Kommandozeile aufnimmt und für diese Datei die Anzahl der Zeichen, Wörter und Zeilen ausgibt.

3.3 Felder, Zeiger und dynamische Speicherverwaltung

Bisher haben wir alle Variablen und Objekte nur eindimensional verwendet. Von jedem Objekt hatten wir immer nur ein Exemplar – und wenn es mal zwei waren, hatten diese unterschiedliche Namen. Dieses Defizit ist aber auf Dauer nicht tragbar. In diesem Abschnitt wollen wir uns endlich mit der Frage beschäftigen, wie man in C++ Felder (Listen oder Vektoren) von Variablen und Objekten aufbaut und somit fast beliebig viel Speicher während der Laufzeit eines Programms nutzbar machen kann.

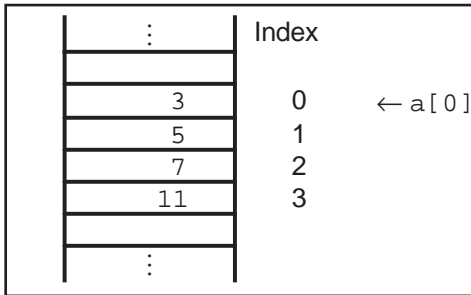
3.3.1 Felder (Arrays)

Deklaration Ein *Feld* (englisch *array*) ist nichts anderes als eine durchnummerierte Menge von Variablen gleichen Typs. Sie können von jedem elementaren und von jedem selbst definierten Datentyp Felder bilden. Dazu geben Sie bei der Deklaration außer Typ und Namen lediglich die Anzahl der Elemente an. Für ein Feld von Ganzzahlen mit 10 Elementen etwa schreiben Sie:

```
int a[10];
```

Größe des reservierten Speicherbereichs Diese Anweisung bedeutet für den Compiler, nicht nur für eine `int`-Variable Speicher zu reservieren, sondern für 10, also insgesamt 40 Bytes – auf einer Intel-32-Bit-Architektur. (Sie können den Speicherverbrauch eines Datentyps oder einer Variablen übrigens auch selbst überprüfen, nämlich mit der Funktion `sizeof()`. In unserem Beispiel liefert `sizeof(int)` den Wert 4 und `sizeof(a)` die Zahl 40.)

Zugriff auf Feldelemente Nun können Sie auf das Feld zugreifen, das heißt seine Elemente mit Werten belegen und später wieder auslesen. Dazu gibt es den *Indexoperator*, der aus einem Paar eckiger Klammern `[]` besteht. In diesen geben Sie den Index des Elements an, das Sie bearbeiten möchten, zum Beispiel:

**Abbildung 3.5**

Ein Feld ist eine Menge gleichartiger Variablen hintereinander.

```
a[0] = 3;
a[1] = 5;
// ...
a[9] = 13;
```

Damit sind wir schon bei der größten Fehlerquelle im Zusammenhang mit Arrays:

Die Indizierung eines Feldes mit n Elementen läuft grundsätzlich von 0 bis $n-1$. Allerdings verhindern weder Compiler noch Laufzeitumgebung, dass Sie auch auf Speicherstellen mit höheren Indizes zugreifen, also etwa in $a[10]$ einen Wert schreiben. Achten Sie also immer darauf, dass Ihre Indizes den zulässigen Bereich $\{0, \dots, n-1\}$ nicht verlassen.

Was geschieht eigentlich, wenn Sie über die Größe Ihres Feldes hinaus schreiben? Das Programm legt zumeist seine Variablen hintereinander an, nach Möglichkeit ohne Lücken. Wenn Sie also beispielsweise nur zehn Elemente reserviert haben und auf ein elftes zugreifen, ändern Sie damit den Wert einer anderen Variablen. Im schlimmsten Fall stehen dort aber Variablen eines anderen Prozesses oder gar Programmanweisungen. Das heißt, Sie ändern den Speicherinhalt an einer kaum vorher-sagbaren Stelle. Entsprechend unvorhersehbar sind die Folgen. Meist führen solche Fehler leider nicht sofort zu Abstürzen, sondern erst einige Zeit später, an einer Stelle mit völlig korrektem Code, der eben auf den zerstörten Speicherbereich zugreifen will. Ihr Programm endet dann abrupt mit der Meldung: *Segmentation fault*. Bei einem solchen Fehler sollten Sie daher immer zuerst an unerlaubte Speicherzugriffe denken.

Folgen von
Falschindizierungen

Ich habe Sie schon bei einfachen Variablen gewarnt, dass diese nach einer Deklaration völlig undefinierte Werte haben können und Sie daher stets so früh wie möglich für eine Initialisierung sorgen sollten. Bei

Initialisierung bei
Deklaration

Feldern vervielfacht sich Ihr Problem lediglich. Aber auch hier können Sie gleichzeitig mit der Deklaration das Feld initialisieren. Dabei geben Sie die gewünschten Inhalte als Liste in geschweiften Klammern { } an, getrennt durch Kommas.

```
int x[3] = {3, 7, 11};
```

*Deklaration ohne
Größenangabe*

Wenn Sie jetzt besonders ökonomisch denken, werden Sie sagen: »Damit gebe ich doch die Anzahl der Elemente zweimal an, einmal als explizite Größenangabe und einmal implizit durch die Zahl der Initialisierungswerte!« In der Tat müssen Sie nämlich für alle deklarierten Elemente auch einen Wert in der Initialisierungsliste eintragen; also könnte man doch diese Anzahl gleich als Größenangabe verwenden. Der Compiler unterstützt solche Überlegungen sehr wohl:

```
int x[] = {3, 7, 11};
```

Ich möchte Ihnen aber empfehlen, derartige Konstrukte nur sehr selten einzusetzen. Ich finde es wesentlich übersichtlicher, wenn man gleich mit einem Blick auf die Deklaration sieht, wie viele Elemente ein Feld hat, das heißt, bis zu welchem Index man zugreifen darf – und nicht erst nachzählen muss. Bei drei Einträgen ist das sicher harmlos, bei circa acht und mehr aber eine zusätzliche Fallgrube. Da es davon ohnehin genug gibt, müssen wir nicht noch selbst welche graben.

*Mehrdimensionale
Felder*

Felder können auch mehrere Dimensionen haben. Dazu fügen Sie einfach weitere Größenangaben in eckigen Klammern an die Deklaration an. Eine 3×4 -Matrix etwa können Sie deklarieren als:

```
double matrix[3][4];
```

Auch beim Zugriff auf die Elemente gilt: pro Dimension ein Index. Wenn Sie diese Matrix beispielsweise mit null initialisieren wollen, brauchen Sie dazu folgende Schleifen:

```
for(int i=0; i<3; i++)
    for(int j=0; j<4; j++)
        matrix[i][j] = 0.0;
```

*Lieber
Standardbibliothek als
Felder!*

Felder nehmen in C++ unter den Top 10 der Quellen der verheerendsten Fehler einen der vordersten Plätze ein. Sie sind ein Relikt aus C, wo man sehr viel Wert auf eine möglichst systemnahe Programmierung legte. In C++ besteht aber eigentlich keine Notwendigkeit dafür, Felder zu verwenden, da die C++-Standardbibliothek alle Arten von Containern, also Listen, Vektoren, Stapel und so weiter, in effizienter und robuster Form bereitstellt. Ich möchte Ihnen daher empfehlen, in Ihren Programmen nach Möglichkeit auf Felder zu verzichten und ausschließlich die

Klassen der Standardbibliothek zu verwenden. Mehr dazu erfahren Sie ab Seite 320.

Ein weiterer Nachteil von Feldern ist, dass Sie bereits im Code die genaue Zahl der Elemente angeben müssen. Der Wert muss dabei auf alle Fälle eine Konstante sein, die während des Kompilierens bestimmbar ist. Es ist also nicht möglich, in dieser Form ein Feld zu definieren, dessen Größe Sie erst während der Laufzeit des Programms ermitteln, zum Beispiel

```
unsigned int n=5;
int a[n]; // compiler error
```

*Nachteil: Feldgröße ist
Kompilierkonstante*

Wie Sie sicher bald merken werden, ist das eine erhebliche Einschränkung. Will man in einem Programm für alle Eventualitäten gewappnet sein, muss man auch mit unerwartet großen Datenfeldern umgehen können. Auch in diesem Punkt bieten Ihnen die Container der C++-Standardbibliothek nur Vorteile.

Hintergrund

Die letzte Aussage muss ich auch gleich wieder relativieren: Nach dem Standard von C und C++ muss die Feldgröße eine Compile-Zeit-Konstante sein. Der GCC verfügt jedoch schon länger über eine Compiler-Erweiterung, die diese Regel außer Kraft setzt. Deshalb können Sie mit ihm auch Funktionen wie die folgende übersetzen:

```
void f(int n)
{
    int a[n];

    // Tue etwas mit a
}
```

*Variable Feldgrößen als
Compiler-Erweiterung*

Ich kann Sie aber nur warnen, eine solche Konstruktion zu nutzen. Denn einer der bedeutenden Vorteile von C und C++ ist die Portabilität, die nur durch die Standardisierung erreicht wurde. Wenn Sie auf Compiler-Erweiterungen vertrauen, bedeutet das auch, dass Ihr Code von kaum einem anderen Compiler noch übersetzbar ist. Außerdem kann diese Erweiterung in künftigen Versionen eventuell wegfallen. Betrachten Sie also lieber stets die Feldgröße als eine feste Konstante, dann bleiben Sie auf der sicheren Seite.

3.3.2 Zeichenketten

Wenn wir bislang Zeichenketten für Namen oder Beschriftungen brauchten, haben wir immer Objekte der Klasse `string` der C++-Standardbibliothek verwendet (Genauerer ab Seite 322). Das ist eine

robuste und sichere Vorgehensweise; allerdings gehört diese Klasse erst seit dem ANSI/ISO-Standard von 1998 verbindlich zu C++. Der traditionelle Weg, Zeichenketten zu speichern, ist der von C übernommene: in Form von Feldern des Typs `char`. Eigentlich bräuchte man sich heute damit gar nicht mehr zu beschäftigen, wenn es nicht viele Systemfunktionen gäbe, die als Argumente oder Rückgabewerte gerade ein solches Zeichen-Array haben. Und da diese Funktionen sämtlich in C geschrieben sind, wird das auch noch länger so bleiben.

Die einfachste Möglichkeit, ein `char`-Feld zu definieren, ist diejenige mit impliziter Größenangabe. Da selten auf einzelne Elemente zuzugreifen ist, wird dieser Weg relativ häufig eingesetzt:

```
char txt[] = "Unser Text.";
```

Nullterminierung

Letztlich ist eine Zeichenkette nur ein Stück Speicher, das dann als Zeichen interpretiert wird. Der Datentyp `char` entspricht nämlich genau einem Byte. Damit das Programm beim Interpretieren weiß, wo der String aufhört und andere Variablen anfangen, haben die Erfinder von C die 0 definiert (so genannte *Nullterminierung*). Das bedeutet, dass

- ❑ jede Zeichenkette als letztes Zeichen den Wert 0 (nicht die Ziffer »0«!) enthält, so dass der Speicherbedarf um eins größer ist als die Anzahl der Zeichen, und
- ❑ beim Fehlen der 0 alle Speicherstellen bis zur nächsten, die den Wert 0 aufweist, als Teil des Strings interpretiert werden.

Letzterer Fall kommt zwar nicht allzu häufig vor, kann aber dann ziemlich unerwartete Ausgaben hervorrufen. Auf der anderen Seite heißt das aber auch, dass Sie den String abkürzen können, wenn Sie eines der Elemente auf 0 setzen. So wird aus obigem String `txt` durch

```
txt[4] = 0;
```

nur noch »Unse«. Folglich ist hier die explizite Größenangabe bei der Deklaration sogar gefährlich, weil Sie dabei leicht die abschließende 0 vergessen.

Eingabe von Zeichen-Arrays

Bei der Eingabe können Sie Zeichen-Arrays wie andere einfache Datentypen behandeln. Dabei ist allerdings die begrenzte Länge zu beachten:

```
char eingabe[20];
cin >> eingabe;
```

Gibt der Benutzer hier mehr als 19 Zeichen ein, kommt es zu einem der gefährlichen Speicherfehler, die ich oben beschrieben habe. Besser ist es, die Methode `getline` zu verwenden, bei der Sie die Maximalgröße der Eingabe zusätzlich übergeben. Ist die Eingabe länger, wird sie abgeschnitten.

```
const int GROESSE = 100;
char eingabe[GROESSE];
cin.getline(eingabe, GROESSE);
```

(Dieser Code lässt sich im Gegensatz zu dem vom Ende des letzten Abschnitts übersetzen, da `GROESSE` als Konstante deklariert ist und sich daher zur Laufzeit nicht ändern kann.)

Wenn Sie auf diese Weise ein Zeichenfeld definiert haben, dürfen Sie ihm nicht als Ganzes ein anderes zuweisen.

```
char txt[] = "Was?";
txt = eingabe; // Fehler
```

Für weitere Bearbeitungsmöglichkeiten von Zeichenfeldern brauchen wir den Begriff des Zeigers, den ich im folgenden Abschnitt einführen werde. Damit können Sie dann Funktionen verwenden, um Zeichenfelder zu kopieren, ihre Länge zu messen, Zeichen darin zu suchen und so weiter. Außerdem werden Sie eine Methode kennen lernen, um Zeichen-Felder mit variabler Größe anzulegen.

3.3.3 Zeiger

In den klassischen C-orientierten Lehrbüchern für C++ taucht der Begriff des Zeigers meist so früh auf, dass er mehr für Verwirrung als für Klarheit sorgt. Dass wir uns durch viele wesentliche Konzepte und Sprachelemente von C++ bis hierher durcharbeiten konnten, ohne Zeiger zu benötigen, macht jedoch deutlich, dass C++ mit einer sehr sparsamen Verwendung von Zeigern auskommt. Diese Vorgehensweise möchte ich Ihnen auch ganz allgemein empfehlen – noch bevor Sie überhaupt wissen, von was da eigentlich die Rede ist.

Was ist ein Zeiger?

Dass wir bislang überhaupt keine Zeiger benötigten, stimmt auch nicht so ganz. Bei einigen Aufrufen von Systemfunktionen habe ich mich nur etwas vor dem Begriff gedrückt und von »Speicherstellen« oder Ähnlichem gesprochen, Sie aber gleichzeitig mit der Syntax etwas im Unklaren gelassen. Wobei die Vorstellung einer »Speicherstelle« dem eigentlichen Begriff aber schon recht nahe kommt.

Definieren wir also:

Ein Zeiger ist eine Variable, die die Speicheradresse einer anderen Variablen (beziehungsweise eines Objekts) enthält.

Abbildung 3.6
Zeiger sind Verweise
auf Speicheradressen.



Sie erfahren über den Zeiger also, an welcher Stelle im Hauptspeicher sich die Variable befindet. Damit ist der Manipulation des Speichers natürlich Tür und Tor geöffnet; entsprechend groß sind die Risiken. Obgleich ein Zeiger immer einen bestimmten Typ haben muss, ist es nicht absolut zwingend, dass der Speicherbereich, auf den er zeigt, ein existierendes Objekt ist. Über den Zeiger kann der Bereich erst als solches interpretiert werden.

Syntax bei Zeigern

Deklaration Man deklariert einen Zeiger auf ein Objekt vom Typ T, indem man den *-Operator hinter den Typ setzt, etwa

```
int* p1;  
double* p2;  
Raumfahrzeug* pUfo;
```

Damit stellt der Zeiger zwar einen eigenen Typ dar, der aber von dem des referenzierten Objekts abhängt. Beachten Sie, dass man auch Zeiger auf einen Zeiger (und so weiter!) definieren kann:

```
char** pp3;
```

Es ist übrigens auch erlaubt, den Stern nicht an den Typ zu hängen, sondern unmittelbar vor die Zeigervariable zu setzen (allerdings auf keinen Fall dahinter!):

```
float *pf;
```

Da der Compiler beide Schreibweisen unterstützt, ist es letztlich Gewohnheits- oder Geschmackssache. Ich finde es besser, den Stern direkt an den Typ zu hängen, weil damit die Bildung des Zeigertyps deutlicher wird.

Wenn Sie mehrere Zeiger gleichzeitig deklarieren, gilt der Stern in dessen nur für den Ersten – oder muss ausdrücklich vor jeden gesetzt werden:



```
// Deklariert einen Zeiger
// und eine int-Variable
int* p1, p2;
// Deklariert zwei Zeiger
int *p3, *p4;
```

Aufgrund dieser Problematik gewinnt die Schreibweise mit dem Stern an der Variablen wieder etwas mehr Sinn.

Die Speicheradresse eines bestehenden Objekts können Sie sich über den *Adressoperator* & verschaffen, beispielsweise:

Der Adressoperator

```
int n;
int* p4 = &n;

Raumfahrzeug Rfz;
Raumfahrzeug* pUfo = &Rfz;
```

Hat man schon einen Zeiger, will man natürlich auch den Inhalt der Speicherstelle, auf die er zeigt, ermitteln und verändern. Dazu wird abermals der Stern verwendet, diesmal als *Dereferenzierungsoperator*.

Dereferenzierung von Zeigern

```
int* pi = &n;
if (*pi < 10)
    *pi = 10;
```

Einen solchen Zugriff nennt man auch *Indirektion*. In diesem Beispiel haben wir den Wert der Variablen n verändert, indem wir den Inhalt ihrer Speicherstelle modifizierten. In diesem Sinne ähneln Zeiger sehr Referenzen; der Unterschied ist, dass bei Referenzen der Compiler bereits für Bestimmung der Adresse und Indirektion sorgt.

Bei Zeigern auf Objekte müssen Sie beachten, dass der ».«-Operator für den Zugriff auf einzelne Elemente Vorrang vor dem Dereferenzierungsoperator hat. Wenn Sie also ein Element verändern möchten, müssen Sie die Indirektion klammern.

Dereferenzierung von Objektelementen

```
Raumfahrzeug* pUfo = &Rfz;
(*pUfo).hoehe = 3890.5;
```

Das ist nicht nur unpraktisch, sondern auch unübersichtlich und fehleranfällig. Daher sollten Sie zur Dereferenzierung von Objektelementen ausschließlich den Operator -> einsetzen. Äquivalent zu obiger Anweisung ist nämlich:

```
pUfo->hoehe = 3890.5;
```

Damit sind auch Methodenaufrufe leichter zu schreiben und zu lesen:

```
bool res = pUfo->setGeschwindigkeit(4000);
```

Zeiger und Arrays

Eine Feldvariable wird in C++ als ein Zeiger auf das erste Feldelement betrachtet (wobei sich der Compiler nur noch für die Initialisierung die Größe merkt). Damit können Sie ein Feld problemlos einem Zeiger zuweisen:

```
int a[6] = {3, 5, 7, 11, 13, 17};  
int* a_ptr = a;
```

Entsprechend können Sie statt über den Indexoperator ein Feld auch wie einen Zeiger dereferenzieren – und umgekehrt. Dabei ist `a[0]` äquivalent zu `*a`, `a[1]` äquivalent zu `*(a+1)`, `a[2]` äquivalent zu `*(a+2)` und so weiter. Da dies auch die Sichtweise des Compilers auf ein Array widerspiegelt, wird Ihnen nun vermutlich etwas klarer, warum eine Überprüfung auf Überschreitung der Feldgrenzen nicht stattfindet.

Nullzeiger

Zeiger sollten ebenso wie alle anderen Variablen unmittelbar bei oder nach der Deklaration initialisiert werden. Ein spezieller Zeigerwert, der für diesen Zweck genutzt werden kann, ist 0. Ein mit 0 belegter Zeiger zeigt definitiv auf »nichts«.

Ebenso sollten Sie einen Zeiger, den Sie momentan nicht benötigen, weil er auf ein noch nicht oder nicht mehr existierendes Objekt verweist, mit 0 belegen.

```
int* iptr = 0;
```

Bevor Sie auf einen Zeiger zugreifen, sollten Sie sicherstellen, dass er auf ein existierendes Objekt zeigt.

```
if (iptr)  
    *iptr = 7;
```

Wenn Sie das nämlich nicht tun und der Zeiger steht auf 0, stürzt Ihr Programm sofort mit einem »Segmentation fault« ab. Dieser Laufzeitfehler wird unter anderem immer dann erzeugt, wenn Sie einen Nullzeiger dereferenzieren wollen.

In C hat man ein Makro namens `NULL` verwendet, um den Nullzeiger darzustellen. Das ist in C++ nicht mehr nötig. Wenn Sie dafür einfach die Zahl 0 benutzen, sind Sie immer auf der sicheren Seite und bekommen zudem mit der Typprüfung weniger Ärger.

Hintergrund

Sie können Zeiger nicht nur statisch verwenden, sondern mit diesen auch rechnen. Dabei spielt der Typ eine sehr große Rolle: Wenn Sie einen Zeiger (um eins) inkrementieren oder dekrementieren, zeigt er anschließend nicht auf das nächste Byte im Speicher, *sondern auf die Adresse der nächsten Variablen desselben Typs*. Was heißt das genau? Der Zeiger wird um so viele Byte verändert, wie ein Objekt des Basistyps beansprucht. (Und das kann für ein und denselben Typ sogar von der Architektur des Rechners abhängen.) Versuchen Sie es mit folgendem Beispiel:

Zeigerarithmetik

```
double d[10];
double *dp1 = d;
double *dp2 = dp1 + 1 ;

cout << "dp2 - dp1: " << dp2 - dp1 << endl;
cout << "int(dp2) - int(dp1): "
    << int(dp2) - int(dp1) << endl;
```

Welche Ausgabe würden Sie erwarten? Das Programm druckt für die erste Zeile 1 und für die zweite 8. Wenn Sie anschließend den Zeiger `dp2` weiter erhöhen, etwa

```
dp2 += 3;
```

und dieselben Druckanweisungen anfügen, erhalten Sie 4 und 32.

Wenn Sie direkt auf einzelne Bytes zugreifen müssen, sollten Sie als Zeigertyp `char*` wählen. Dieser ist auf die Größe 1 Byte festgelegt, so dass Sie damit wirklich Byte für Byte erfassen. Sie sehen daran schon, dass Zeigerarithmetik zur systemnahen Programmierung gehört und als solches eine Spezialität der Programmiersprache C darstellt. Unser C++ hat dessen Fähigkeiten geerbt, was aber noch nicht heißen soll, dass dieser Stil für Sie die Regel werden sollte.

Ein beliebtes Beispiel für eine Anwendung der Zeigerarithmetik ist das Umkopieren eines Zeichenfeldes.

```
char quelle[] = "Hier sind die Daten zuhause.";
char ziel[50];

char* p= quelle, *q= ziel;
while (*q++ = *p++);
```

So elegant das für manche auch aussehen mag: ich finde hier gleich mehrere Punkte, die man besser nicht so machen sollte. Sie auch? Besonders raffinierte C-Hacker packen die beiden letzten Zeilen gleich in eine:

```
for(char* p= quelle, *q= ziel; *q++ = *p++; );
```

Wie lange brauchen Sie, um zu verstehen, was hier passiert? Für mich ist es einfach schlechter Stil, wenn ein Programmcode die Problemlösung, die mit ihm eigentlich erreicht werden soll, eher verschleiert als verdeutlicht. Wenn Sie jetzt versuchen, obigen Codeteil so umzuschreiben, dass nur die Ausgangsvariablen `quelle` und `ziel` als Arrays verwendet werden und eine `while`-Schleife beziehungsweise eine `for`-Schleife zur Steuerung eingesetzt wird, werden Sie merken, wie man das Ganze auch übersichtlich und auf den ersten Blick verständlich programmieren kann.

3.3.4 Dynamische Speicherverwaltung

Bereiche des Speichers

Wenn Sie ein Programm starten, wird es in den Speicher geladen. Es belegt aber gleich von Anfang an noch zusätzlichen Hauptspeicher. Neben dem Code werden zwei weitere Bereiche reserviert:

- ❑ einer für die *statischen Variablen* (globale Variablen und Klassenattribute, die zu Programmstart schon definiert sind) und
- ❑ ein so genannter *Stack*, auf dem die lokalen Variablen und die Funktionsparameter abgelegt werden.

Die Größe des Stack wird beim Programmstart festgelegt und ist anschließend nicht mehr zu ändern. Wenn Sie viele Variablen oder sehr große Felder anlegen, kann es da schon eng werden. Aber eigentlich ist der Hauptspeicher bei den heutigen Rechnern im Allgemeinen sehr viel größer und meist nicht komplett belegt. Wenn der Stack schon nicht mehr hergibt, wie kommen wir sonst an den Rest des Hauptspeichers heran?

Der Heap

Aus Sicht des Programms bezeichnet man den gesamten restlichen freien Speicher der Maschine als *Heap*. Auf ihm können Sie Objekte und Felder dynamisch anlegen (also während der Laufzeit), wobei Sie nur die physikalische Speichergröße beschränkt.

Gegenüberstellung

Stack – Heap

Halten wir also fest: Der Stack ist der Teil des Arbeitsspeichers, der beim Start des Programms dafür frei gehalten wird und der alle lokalen Variablen sowie die Funktionsparameter enthält. Der Heap umfasst das gesamte restliche freie RAM und enthält die dynamisch angelegten Objekte und Felder. Die Vor- und Nachteile sind dabei:

Stack	Heap
+ schnelle Reservierung	+ sehr große Speichermenge zur Verfügung
- feste Größe	- Verwaltung schafft etwas Overhead - Der Programmierer muss den benötigten Speicher selbst reservieren und freigeben (⇒ Fehlerquelle!)

Das Anlegen von Objekten auf dem Heap hat aber auch noch einen weiteren Vorteil: Sie können Objekte über den Gültigkeitsbereich einer Funktion oder Klasse hinaus weiterleben lassen. Da Sie die dynamisch erzeugten Objekte selbst wieder freigeben müssen, heißt das natürlich auch, dass diese so lange existieren, bis sie ausdrücklich freigegeben werden – eben auch über das Ende einer Funktion hinaus.

*Dynamische Objekte
haben eigenen
Gültigkeitsbereich*

Wann sollte man also was nehmen? Immer, wenn Sie beim Schreiben des Programms genau wissen, wie groß ein Feld sein wird, können Sie dieses statisch reservieren – sofern nicht allzu viele davon bei Ihnen angelegt werden müssen. In den meisten Fällen ist es aber leider so, dass die Größe im Voraus nicht genau bestimmbar ist. Dann bleibt Ihnen nichts anderes übrig, als Ihr Array auf dem Heap anzulegen. Dasselbe gilt, wenn es sich um sehr viele Elemente handelt. Es ist jedoch sinnvoll, den Umgang mit Feldern in Klassen zu kapseln, damit nicht jeder Programmabschnitt mit dynamischem Speicher hantieren muss, sondern das von den Methoden der entsprechenden Klasse komplett erledigt wird.

Wann auf den Heap?

Dynamisches Reservieren von Speicher mit new

Um Speicher auf dem Heap zu reservieren (zu *allozieren*, wie man sagt), gibt es den Operator `new`. Dieser belegt genau so viel Speicher, wie das Objekt tatsächlich benötigt. Die Syntax lautet allgemein:

```
Typ* t = new Typ;
```

Wollen wir also ein Objekt `Raumfahrzeug` anlegen und damit arbeiten, können wir schreiben:

```
Raumfahrzeug* r = 0; //Zeiger initialisieren
r = new Raumfahrzeug; //Speicher reservieren
//Objekt verwenden
r->setGeschwindigkeit(20000);
```

Zugriff vorwiegend
über Zeiger

Dieses Beispiel zeigt auch, weshalb ich erst an dieser Stelle auf dynamische Objekte zu sprechen komme: Der Zugriff erfolgt nämlich vorwiegend über Zeiger. Was ist dabei mit »vorwiegend« gemeint? Zum Anlegen (und späteren Löschen) sowie zum Zugriff auf Attribute und Methoden braucht man einen Zeiger auf das Objekt. Allerdings können Sie den Zeiger auch an eine Funktion übergeben, die eine Referenz dieses Objekts erwartet. Dazu müssen Sie ihn natürlich bei der Übergabe dereferenzieren. Die Funktion kann dann mit der Referenz wie gewohnt arbeiten; sie merkt also nicht, dass es sich eigentlich um ein dynamisch verwaltetes Objekt handelt.

Konstruktoraufruf bei
new

Sie können mit `new` sowohl Variablen von einfachen Datentypen als auch von selbst definierten Strukturen und Klassen anlegen. (Einzelne Variablen vom Typ `int` oder `float` dynamisch anzulegen, ist jedoch ziemlich unüblich.) Bei Objekten kommt noch eine Besonderheit hinzu: der Konstruktor. Wenn Sie Instanzen auf dem Stack anlegen, können Sie ja durch die Angabe von Argumenten einen überladenen Konstruktor aufrufen. Das ist bei `new` genauso möglich. Erinnern Sie sich noch an die Klasse `Datum`, die wir in Abschnitt 2.6 als Beispiel erstellt haben? Dort gab es etwa die Konstruktoren

```
class Datum
{
public:
    Datum();
    Datum(unsigned int _t,
           unsigned int _m, unsigned int _j);
    // ...
};
```

Wenn wir Objekte dieses Typs dynamisch anlegen, können wir beispielsweise schreiben:

```
int main()
{
    // Standardkonstruktor
    Datum* pHeute = new Datum;

    // Konstruktor mit 3 Argumenten
    Datum* pOstern = new Datum(4,4,1999);
    // ...
}
```

Freigeben von dynamisch angelegten Objekten mit `delete`

Alle Objekte, die Sie mit `new` angelegt haben, müssen Sie auch selbst wieder freigeben! So simpel diese Regel klingt, so wichtig ist es doch, sie

zu beherzigen. Denn einige Probleme, die Programme mit dynamischer Speicherverwaltung immer wieder haben, sind »verwaiste Speicherbereiche«, also allozierter Speicher, auf den keiner mehr zugreifen kann.

Der Operator zum Freigeben heißt `delete`. Als Argument dahinter müssen Sie einen Zeiger auf den reservierten Bereich angeben (also genau die Adresse, die Sie von `new` als Rückgabewert bekommen haben). Ebenso wie bei `new` ein Konstruktor aufgerufen wird, findet bei `delete` ein Aufruf des Destruktors statt.

```
Raumfahrzeug* r = 0; //Zeiger initialisieren
r = new Raumfahrzeug; //Speicher reservieren
// .. Objekt verwenden
delete r; //Speicher freigeben
```

Das bedeutet auch, dass Sie darauf achten müssen, sich die Adresse so lange zu merken, bis Sie den Speicher freigeben wollen. Dynamisch angelegter Speicher kann nämlich auch dann noch alloziert sein, wenn die Variable, über die er einst angelegt und angesprochen wurde, längst nicht mehr existiert. Da also die Speicherreservierung unabhängig von allen Gültigkeitsbereichen von Blöcken, Funktionen und so weiter ist, sind Sie selbst dafür verantwortlich, die Adresse zum Zeitpunkt der Freigabe noch verfügbar zu haben.

Neuralgische Punkte bei der dynamischen Speicherverwaltung

Das Arbeiten mit dynamisch verwaltetem Speicher ist zwar in den meisten größeren Programmen unumgänglich, stellt aber auch eine Hauptfehlerquelle bei der Programmentwicklung mit C++ dar. Im Gegensatz zu einigen anderen Programmiersprachen (wie Java oder Fortran) ist C++ in diesem Punkt sogar besonders sensibel. Aus diesem Grund gibt es auch eine Reihe von kommerziell erhältlichen Werkzeugen – auch unter Linux –, deren Hauptaufgabe es ist, Fehler bei der Speicherverwaltung ausfindig zu machen (*Insure++* von ParaSoft ist beispielsweise in einer Demoversion Teil der SuSE-Distribution oder unter www.parasoft.com/products/insure zu finden). Der Programmierer ist gut beraten, wenn er sich bereits beim Schreiben des Codes darüber Gedanken macht und so mögliche Fehlerquellen von vornherein ausschließt. Darüber hinaus bietet gerade C++ mit seinem Konzept der Daten- und Prozessabstraktion die Möglichkeit, dynamische Speicherverwaltung an einigen wenigen Stellen im Gesamtprogramm zu kapseln und so die Gefahr, dass dabei Fehler unterlaufen, relativ klein zu halten.

Im Folgenden will ich Ihnen einige neuralgische Punkte vorstellen, auf die Sie bei der Programmierung besonders achten sollten. Da dieses Thema allein schon ganze Bücher füllt (zum Beispiel [BOEHM 2000])

und die sehr empfehlenswerten [MEYERS 1996], [MEYERS 1997]), kann hier selbstverständlich kein Anspruch auf Vollständigkeit erhoben werden.

Kein Speicher verfügbar

Zunächst ist zu bedenken, dass es auch bei `new` selbst Probleme geben kann. Auch die heutigen RAM-Größen stoßen irgendwann an ihre Grenzen. Der ANSI/ISO-Standard sieht vor, dass der `new`-Operator eine so genannte Ausnahme (namens `bad_alloc`, siehe Seite 366) auslösen soll. Bei der vorherigen Version (2.95) des GCC war dies bereits eingebaut, aber noch auskommentiert (etwa in der Datei `stl_alloc.h`). Ab Version 3.0 wird dieses Verhalten unterstützt – jedoch nur, wenn Sie die entsprechende Header-Datei über

```
#include <new>
```

einbinden. Standardmäßig zeigt der Compiler noch das aus C bekannte Verhalten: Ist nicht genügend Speicher verfügbar, gibt `new` den Zeigerwert 0 zurück. Sie können daher, wenn Sie Ihr Programm robust machen wollen, nach jeder Speicherreservierung die Rückgabe überprüfen.

```
Raumfahrzeug* r = new Raumfahrzeug;
if (!r) {
    cerr << "Kein Speicher verfügbar!" << endl;
    exit(1);
}
```

Ob Sie in diesem Fall Ihre Anwendung gleich ganz beenden oder versuchen, noch ein Stück weiter zu kommen, bleibt Ihnen überlassen. Erfahrungsgemäß können Sie aber ohnehin nicht mehr viel machen, wenn kein Speicher mehr zur Verfügung steht. Da Linux zudem mit Swap-Speicher arbeitet, das heißt die physikalische RAM-Größe um einen Bereich der Festplatte erweitert, kann es vorkommen, dass bei Speicherknappheit die Swap-Aktivitäten so umfangreich werden, dass das Betriebssystem ausgelastet ist, noch bevor ein `new`-Operator in Ihrem Programm eine Null zurückgibt.

new ohne delete

Weiterhin müssen Sie darauf achten, dass Sie für jeden Bereich, den Sie mit `new` anlegen, ein korrespondierendes `delete` aufrufen. Wie ich Ihnen oben schon erklärt habe, kann ein Versäumnis zu nicht mehr zugänglichen Speicherbereichen führen.

delete ohne new

Außerdem dürfen Sie kein `delete` auf Speicher aufrufen, den Sie auf dem Stack, also *ohne new* angelegt haben.

```
Datum d;
Datum* pD = &d;
// ...
delete pD; //Absturz!
```

Einen solchen Versuch quittiert Linux unweigerlich mit einem *Segmentation fault*.

Ähnliches widerfährt Ihnen, wenn Sie versuchen, ein Objekt mehrmals zu löschen. »Doppelt gelöscht ist endgültig weg« gilt nämlich ganz und gar nicht.

Doppeltes Löschen

```
Datum* pD = new Datum;
// ...
delete pD;

// ...
delete pD; //Absturz!
```

Es ist daher empfehlenswert, nach dem Löschen eines Objekts den Zeiger sofort auf 0 zu setzen. Denn wenn Sie `delete` auf einen Nullzeiger anwenden, ist das völlig ohne Wirkung und daher unkritisch.

Schließlich möchte ich Sie nochmals auf das oben bereits diskutierte Problem hinweisen, dass Speicher, der mit `new` angelegt wurde, auch über das Ende des aktuellen Blocks beziehungsweise der aktuellen Funktion hinaus reserviert bleibt. Sollten Sie den Zeiger darauf »verlieren«, entsteht ein allozierter Speicher, auf den Sie keinerlei Zugriff mehr haben. Im Angelsächsischen nennt man das ein *memory leak*, ein Speicherleck.

Speicherlöcher

Dynamisch erzeugte Felder

Mit `new` können Sie nicht nur einzelne Objekte erzeugen, sondern auch Felder (fast) beliebiger Größe. Im Unterschied zu den auf dem Stack angelegten Arrays haben Sie bei dynamisch verwalteten die Möglichkeit, die Anzahl der Objekte erst zur Laufzeit festzulegen. Sie können daher Ihre Felder exakt an das Problem anpassen, das Sie mit Ihrem Programm gerade behandeln wollen. Die Syntax ist der von gewöhnlichen Feldern ganz ähnlich:

```
cout << "Größe angeben: ";
int groesse = 0;
cin >> groesse;
int* pVector = new int[groesse];
```

Beachten Sie, dass Sie alle Felder, die Sie auf diese Weise angelegt haben, mit einer Variante von `delete` freigeben müssen, die ein Paar eckiger Klammern `[]` als Hinweis auf die Array-Eigenschaft enthält.

*Freigabe mit
delete[]!*

```
delete[] pVector;
```

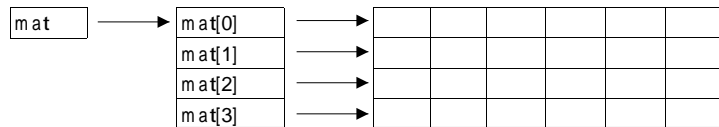
Leider weisen Sie weder Compiler noch Laufzeitumgebung auf den Fehler hin, wenn Sie nur das einfache `delete` verwenden. Umso wichtiger ist es daher, dass Sie sich selbst über den Typ Ihrer Objekte im Klaren sind. Wenn Sie nämlich nur das einfache `delete` auf ein Feld anwenden, geben Sie damit nur das erste Element frei und lassen die anderen alloziert – aber völlig unzugänglich!

Mehrdimensionale
Felder

Ebenso leicht können Sie auch mehrdimensionale Felder anlegen. Die einfache Vorgehensweise dabei ist, zunächst ein Feld von Zeigern auf die Arrays zu definieren, die eine Dimension weniger haben. Die Felder mit Dimension eins werden dann angelegt wie oben beschrieben.

Abbildung 3.7

Eine Matrix ist ein
Zeiger auf ein Feld von
Zeigern.



Für eine Matrix, also ein zweidimensionales Feld, hat das etwa folgende Form:

```

int z, s;
cout << "Zeilen und Spalten eingeben: ";
cin >> z >> s;

int** mat = new int*[z];
for(int i=0; i<z; i++)
    mat[i] = new int[s];
  
```

Ist ein solches Feld einmal angelegt, können Sie darauf genauso zugreifen, als wäre es statisch erzeugt:

```
mat[i][j] = 12;
```

Zum Freigeben müssen Sie in der umgekehrten Reihenfolge vorgehen:

```

// Zeilen freigeben
for(int i=0; i<z; i++)
    delete[] mat[i];

// Feld mit Zeigern auf Zeilenanfänge freigeben
delete[] mat;
  
```

Konstruktoraufrufe bei
Feldern

Wir hatten oben festgestellt, dass bei Objekten mit jedem `new` auch

der Konstruktor aufgerufen wird. Dabei ist sogar die Angabe von Argumenten zur Verzweigung zu einem überladenen Konstruktor möglich. Bei Feldern von Objekten erlaubt C++ die Verwendung allgemeiner Konstruktoren indessen nicht. Für alle Objekte des Feldes wird lediglich der Standardkonstruktor ausgeführt. Wenn Sie den Objekten zusätzliche Informationen mitgeben wollen, müssen Sie dies über eine Methode, zum Beispiel mit Namen `init()`, machen, die Sie im Anschluss an die Erzeugung aufrufen.

3.3.5 Konstruktoren und Destruktoren

Unter dem Blickwinkel der dynamischen Speicherverwaltung bekommen auch bereits besprochene Sprachelemente neue Bedeutung. Wenn eine Klasse dynamisch angelegte Speicherbereiche verwaltet, sollten Sie bei ihrer Erzeugung und Vernichtung besondere Sorgfalt walten lassen. Betrachten wir dazu eine Klasse `Vektor`, die einen beliebig langen Vektor von Gleitkommazahlen doppelter Genauigkeit darstellen soll. Die Deklaration lautet etwa:

```
class Vektor
{
private:
    unsigned int size;
    double* v;

public:
    Vektor();
    Vektor(unsigned int _size);
    Vektor(Vektor& _vek);
    ~Vektor();
    unsigned int getSize();
    const double& at(unsigned int _i) const;
    double& at(unsigned int _i);
};
```

Wie Sie sehen, besteht die Klasse im geschützten Teil aus einem `unsigned int`-Attribut für die Anzahl der Elemente und einem Zeiger auf das Feld. Der öffentliche Teil enthält drei Konstruktoren, einen Destruktor und zwei Zugriffsmethoden. Diese decken das gesamte Spektrum der Anwendungsfälle ab. Die konstante Variante wird verwendet, wenn reine Lesezugriffe benötigt werden; dabei kann sogar das Objekt konstant deklariert sein. Die andere Version benutzt man bei Schreibzugriffen. Hier haben Sie übrigens den einzigen Fall vor sich, wo eine Methode mit einer anderen überladen werden kann, die *denselben Namen und dieselbe Signatur* hat und sich nur durch die Auszeichnung als

Die Zugriffsmethoden

konstante Methode unterscheidet. Wie diese Methoden zu implementieren sind, überlasse ich Ihnen (zur Übung).

Doch nun zum interessanten Teil:

Kopierkonstruktoren

Die beiden ersten Konstruktoren sind recht einfach. Der Standardkonstruktor initialisiert alle Datenelemente mit 0, während der Ganzzahlkonstruktor einen Vektor der angegebenen Größe anlegt.

```
Vektor::Vektor() :
    size(0), v(0) {};

Vektor::Vektor(int _size) :
    size(_size)
{
    v = new double[_size];
}
```

*Kopieren ohne
Kopierkonstruktor*

Wozu braucht man nun einen Kopierkonstruktor? Zur Beantwortung dieser Frage sollten wir uns zunächst überlegen, was denn passiert, wenn wir auf einen solchen verzichten. Der Compiler sorgt dafür, dass bei der Erzeugung eines neuen Objekts aus einem bestehenden alle Datenelemente bitweise eins zu eins kopiert werden. Für Zeiger bedeutet das, dass auch das neue Objekt *auf denselben Speicherbereich* zeigt wie das vorhandene. Beide Vektor-Objekte sind damit nicht mehr unabhängig voneinander verwendbar, da sie einen gemeinsamen Speicherbereich referenzieren. Schreibzugriffe auf das eine Objekt wirken sich auch auf das andere aus. Die Situation eskaliert, wenn etwa das erste Objekt vernichtet wird. Damit wird nämlich auch sein Feld freigegeben, so dass das zweite auf einen völlig undefinierten Speicherbereich zugreifen würde – mit baldigem Totalabsturz.

Um das zu vermeiden, müssen Sie bei Klassen, die mit dynamisch erzeugtem Speicher arbeiten, stets Kopierkonstruktoren selbst schreiben. In diesen können Sie dann dafür Sorge tragen, dass der Inhalt so kopiert wird, wie man es erwarten würde. In unserem Beispiel hieße das, ein neues Feld anzulegen und nur die Werte des alten dorthin zu übertragen.

```
Vektor::Vektor(Vektor& _vek) :
    size(_vek.size)
{
    v = new double[_size];
    for(unsigned int i=0; i<size; i++)
        v[i] = _vek.v[i];
}
```

(In diesem Zusammenhang noch eine kleine Quizfrage: Warum dürfen wir hier auf das eigentlich geschützte Attribut `v` von `vek` zugreifen? Weil eine Klasse immer mit anderen Objekten derselben Klasse befreundet ist.)

Für Zuweisungen zwischen zwei bestehenden Objekten gelten diese Ausführungen übrigens auch. Wie man dazu den Zuweisungsoperator überlädt, erfahren Sie in einem späteren Kapitel. Schon jetzt aber sollten Sie sich als Faustregel merken: Wo immer Sie einen Kopierkonstruktor brauchen, benötigen Sie fast immer auch einen Zuweisungsoperator.

*Ergänzung:
Zuweisungsoperator*

Destruktor

Jeden Speicherbereich, den Sie mit `new` reserviert haben, sollten Sie auch wieder freigeben. In der C-Programmierung hat dies häufig Probleme bereitet, denn bei lokal allozierten Feldern muss eine passende Freigabe bei jedem Rücksprung mit `return` eingebaut werden. Die Arbeit mit Objekten macht da vieles einfacher, denn ein Objekt weiß selbst, was zu tun ist, wenn es vernichtet werden soll. Sie als Programmierer können das Verhalten in dieser Situation über den Destruktor bestimmen.

In unserem Fall wollen wir natürlich, dass der reservierte Bereich freigegeben wird – vorausgesetzt, es gab überhaupt eine Reservierung. Das können wir daran erkennen, dass der Zeiger `v` einen anderen Wert als 0 aufweist, mit dem wir ihn ja im Standardkonstruktor vorbelegt haben.

```
Vektor::~~Vektor()
{
    if (v)
        delete[] v;
}
```

Damit ist erreicht, dass der vom Objekt angelegte Speicherbereich wieder freigegeben wird, wann immer das Objekt seine Gültigkeit verliert.

```
int lies_vektor()
{
    int groesse;
    ifstream eingdatei("eingabe.dat");
    eingdatei >> groesse;
    Vektor v(groesse); // Vektor anlegen
    for(int i=0; i<groesse; i++)
    {
        eingdatei >> v.at(i);
        if (eingdatei.eof())
```

```

    {
        cerr << "Unerwartetes Dateiende!" << endl;
        return -1;
    } // Hier wird v.~Vektor() aufgerufen!
}
// ...
}

```

Sie sehen an diesem Beispiel, dass durch die Freigabe im Destruktor auch bei unerwarteten Rücksprüngen ein ordnungsgemäßes Zerstören des Objekts `v` gewährleistet ist.

3.3.6 Beispiel: CGI-Programmierung

Wer sich heute mit der Entwicklung von System- oder Anwendungssoftware beschäftigt, kennt immer auch zumindest die Grundlagen der Gestaltung von HTML-Seiten für Webserver. Überhaupt ist die Web-Programmierung einer der am schnellsten wachsenden Bereiche in der IT-Entwicklung. Der Siegeszug des Internets ist dabei allgegenwärtig.

Interaktivität im Web

Wenn Sie sich schon intensiver mit der Erstellung von Webseiten beschäftigt haben, werden Sie schnell auf ein grundlegendes Defizit von HTML gestoßen sein: Informationen auf HTML-Seiten sind von Haus aus statisch; sie werden beim Benutzer nur präsentiert, ohne dass dieser irgendwelche Einflussmöglichkeiten darauf hätte. Oftmals wollen Sie als Web-Autor aber gerade, dass der Betrachter mit Ihrem Server interagieren kann, etwa um seine Meinung zu hinterlassen, eine Anfrage oder Bestellung aufzugeben oder nur um nach einem Stichwort zu suchen. In diesem Abschnitt wollen wir uns folglich mit der Frage beschäftigen, wie wir das Senden von Informationen vom (Web-)Client zum Server realisieren können und wie der Server dynamisch auf die Anfragen reagieren kann.

Das Common Gateway Interface

Um diesem Problem zu begegnen, hat man schon sehr früh eine Programmierschnittstelle geschaffen, das *Common Gateway Interface*, kurz *CGI*. Darüber erstellte Programme laufen auf dem Server und sorgen für die Interaktion mit dem Benutzer, das heißt, sie verarbeiten die von ihm eingegebenen Informationen und senden ein für ihn speziell erzeugtes Antwortdokument zurück (Abbildung 3.8). Über diese Schnittstelle können Sie im Grunde alle Arten von Programmen an das Web anbinden, die nur über eine einfache Eingabe verfügen und die in relativ kurzer Zeit ein für den Benutzer interessantes Resultat hervorbringen. Denn als Web-Autor müssen Sie sich verschiedenen Herausforderungen in Bezug auf interaktive Webseiten stellen:

- ❑ Auf der einen Seite befindet sich der Benutzer, der nicht gewillt ist, allzu lange auf das Ergebnis seiner Anfrage zu warten.
- ❑ Auf der anderen Seite steht Ihr Server, der in der Lage sein muss, auch eine große Zahl von Benutzeranfragen gleichzeitig zu behandeln – wenn Ihre Seiten entsprechend attraktiv sind.

Noch vor wenigen Jahren war es üblich, CGI-Programme als Skripten in der Programmiersprache Perl zu erstellen. Für viele kleinere Aufgaben (wie Registrierung eines Benutzers) ist das sicher eine probate Methode. Als das Internet ein immer größeres Publikum anzog, schufen die Serverhersteller Netscape und Microsoft mit NSAPI beziehungsweise ISAPI proprietäre Schnittstellen, die dem Web-Programmierer zusätzliche Möglichkeiten bieten. Mittlerweile sind noch andere Technologien wie Java Server Pages (JSP) oder PHP hinzu gekommen.

Entwicklung der CGI-Programmierung

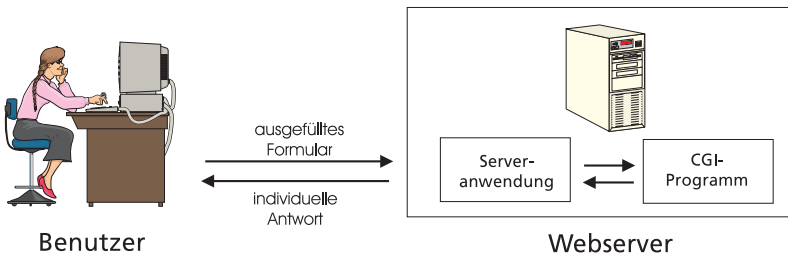


Abbildung 3.8
 Ein CGI-Programm verarbeitet Benutzereingaben und schickt individuelle Antwortdokumente.

Dabei haben jedoch die CGI-Interfaces nach meiner Meinung keineswegs ausgedient. Nicht jeder betreut ja eine Website, auf der er mit einer Million Zugriffen und mehr am Tag rechnen muss. Gerade für Web-Autoren, die die Betreuung in ihrer Freizeit erledigen, ist CGI aufgrund seiner Einfachheit immer noch aktuell. Und auch wenn Perl dafür weit verbreitet ist, sollten wir bedenken, dass es nach wie vor möglich ist, ein ausführbares Programm über CGI aufzurufen – das beispielsweise in C++ geschrieben wurde. Ein solches wollen wir gleich erstellen.

Der Apache-Webserver

Der weltweit meistverbreitete Webserver ist *Apache*. Über 60% der Server im Internet setzen ihn ein. Auch er ist freie Software und im Quelltext erhältlich (über www.apache.org). Neben vielen anderen Plattformen ist er natürlich auch unter Linux verfügbar und dort in den meisten Distributionen enthalten. Möglicherweise wurde er aber nicht standardmäßig bei Ihnen installiert, so dass Sie ihn erst von der CD auf Ihr System kopieren müssen.

*Dateiorganisation bei
Apache*

Die Dateien sind bei Apache standardmäßig wie folgt organisiert:

- ❑ Die für den Webserver relevanten Dateien wie HTML-Files, CGI-Programme, Bitmaps und so weiter sind in einem Systemverzeichnis abgelegt. Über dessen genauen Ort gibt es noch keine allgemein gültigen Richtlinien, so dass die Vorgabeeinstellung dafür sich von einer Distribution zur anderen unterscheiden kann. Bei Red Hat ist es beispielsweise `/home/httpd`, bei SuSE `/usr/local/httpd`. Sie können es aber auch an einen ganz anderen Ort legen. Denn diese und andere Eigenschaften des Apache-Servers sind Teil der voreingestellten Konfiguration, die in der Datei `/etc/httpd.conf` gespeichert ist. Wenn Sie also eine andere Einstellung bevorzugen, müssen Sie lediglich diese Konfigurationsdatei entsprechend anpassen. Im Hauptverzeichnis gibt es verschiedene Unterverzeichnisse, unter anderem:
 - ❑ `htdocs` (oder `html`) enthält die HTML-Dateien. Aus Sicht des Browsers ist dies das Hauptverzeichnis des Servers. Wenn Sie zum Beispiel Ihren Server auf dem Rechner `goethe.abc.de` betreiben, entspricht der URL `http://goethe.abc.de/index.html` die Datei `htdocs/index.html`. Unter `htdocs/manual` befindet sich übrigens ein ausführliches Handbuch zu Apache.
 - ❑ `cgi-bin` enthält die CGI-Programme und -Skripten. Da es sich außerhalb der über URL erreichbaren Hierarchie befindet, müssen die Programme auf anderem Weg aufgerufen werden. Doch dazu gleich mehr.
- ❑ Wenn der Server eine Anfrage eines Browsers nicht korrekt beantworten kann oder ein Serverprogramm fehlschlägt, erzeugt der Server eine Fehlermeldung in der Datei `httpd.error_log` oder `error.log`. Diese können Sie meist im Verzeichnis `/var/log` finden.
- ❑ Die ausführbare Datei des Servers steht wahrscheinlich bei Ihnen unter `/usr/sbin` oder `/bin` und trägt den Namen `httpd`. Sicher hat Ihnen Ihr Installationsprogramm auch ein Skript eingerichtet, um den Server hoch- und herunterzufahren.

Mehr kann ich in diesem Rahmen leider nicht ausführen. Für tiefer gehende Informationen werfen Sie am besten einen Blick in die Apache-Dokumentation oder in eines der vielen Online-Tutorials im Internet.

Formulare

Das Mittel, um dem Benutzer Eingaben auf HTML-Seiten zu erlauben, ist das `<FORM>`-Tag. Mit dessen Hilfe kann der Browser Steuerelemente wie Eingabefelder oder Listfelder darstellen. Auch dazu kann ich Ihnen hier leider nur ein paar Stichworte erklären. Grundkenntnisse in HTML kann ich bei Ihnen ja sicher voraussetzen.

Das Element `<FORM>` umschließt die Angabe der Steuerelemente. Es trägt selbst zwei Attribute (ein drittes optionales lasse ich hier weg):

Das `<FORM>`-Tag

- Mit `ACTION` geben Sie den Pfad oder die URL des Programms an, das die Eingaben verarbeiten soll.
- Über `METHOD` spezifizieren Sie die Art der Übertragung, nämlich als `GET` oder als `POST`. Die beiden Typen unterscheiden die Art, wie die Informationen auf dem Server bereitgestellt werden.
 - Bei `GET` werden die eingegebenen Daten als Kommandozeilenoptionen an das aufgerufene Programm übergeben. Da dies in manchen Fällen Probleme bereitet, wird die Kommandozeile zusätzlich in der Umgebungsvariablen `QUERY_STRING` hinterlegt. `GET` wird üblicherweise für einfachere Formulare verwendet.
 - Bei `POST` erhält das Programm alle Daten über den Standardeingabekanal. Dieser Weg ist hauptsächlich für umfangreichere Formulare gedacht. Im Allgemeinen steht Ihnen aber frei, welche Methode Sie verwenden.

Für Eingabeelemente gibt es eine Reihe von Tags. Neben `<SELECT>` für Listfelder und `<TEXTAREA>` für mehrzeilige Textfelder verwendet man vor allem das `<INPUT>`-Element. Seine Funktionalität und sein Erscheinungsbild können Sie über das Attribut `TYPE` festlegen, wobei `TEXT` der Vorgabewert ist und einem Eingabefeld entspricht; es sind aber auch `CHECKBOX` für Kontrollkästchen oder `RADIO` für Radio-knöpfe möglich. Die Schaltflächen zum Abschicken der Eingaben beschreiben Sie mit `INPUT`; der zugehörige Typ heißt `SUBMIT`. Im Attribut `VALUE`, das sonst für Vorgabewerte dient, steht dann der Text der Schaltfläche.

Tags für Eingabelemente

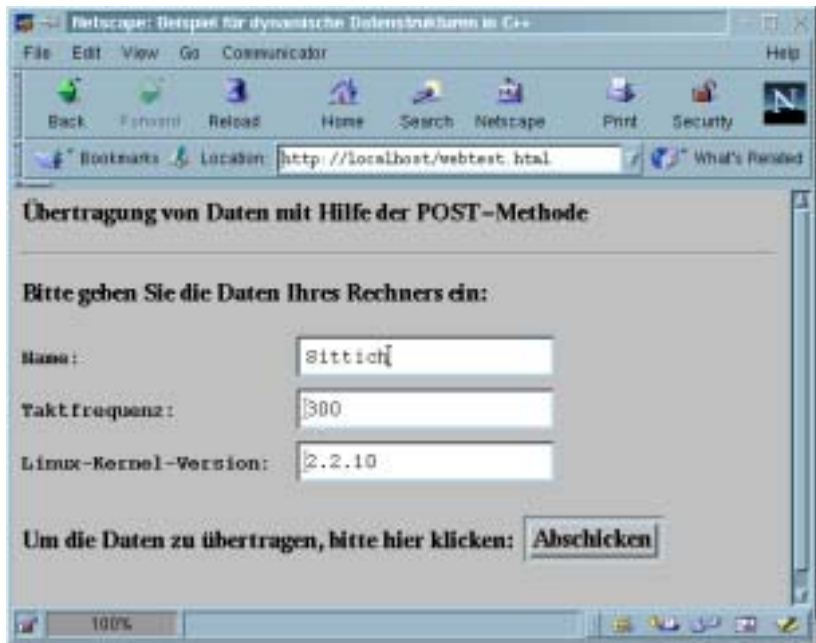
Sehen wir uns ein Beispiel an:

```
<H1>&Uuml;bertragung von Daten mit
Hilfe der POST-Methode</H1>
<HR>
<FORM METHOD="POST" ACTION="/cgi-bin/webtest">
<H2>Bitte geben Sie die Daten Ihres
Rechners ein:<H2>
```

```
<P>
<PRE>
Name:                               <INPUT NAME="Name"
VALUE="" >
Taktfrequenz:                       <INPUT NAME="Frequenz"
VALUE="300" >
Linux-Kernel-Version: <INPUT NAME="Kernel"
VALUE="2.4.2" >
</PRE>
<P>
Um die Daten zu &uuml;bertragen, bitte
hier klicken:
<INPUT TYPE="submit" VALUE="Abschicken">
</FORM>
```

Das Ergebnis dieses HTML-Codes sehen Sie in Abbildung 3.9. Mehr zur Syntax von HTML und seinen Formularen erfahren Sie beispielsweise bei Stefan Münz unter www.teamone.de/selfhtml.

Abbildung 3.9
Über Formulare wie dieses kann der Benutzer Informationen an den Server senden.



Wir wollen uns nun daran machen, diese Daten auszuwerten.

Informationsübertragung an den Server

Wenn der Benutzer auf die Schaltfläche ABSCHICKEN klickt, verpackt der Browser die auf dem Formular eingegebenen Informationen und sendet sie wie üblich mit Hilfe des HTTP-Protokolls an den Server. Mit dem »Verpacken« sind einige Arbeitsschritte verbunden:

Kodierung der Eingaben

- ❑ Alle Leerzeichen in der Eingabe werden durch »+« ersetzt. Alle Sonderzeichen durch ihren ASCII-Wert in Hexadezimaldarstellung %hh.
- ❑ Die einzelnen Felder des Formulars werden zu Schlüssel/Werte-Paaren zusammengefasst, getrennt durch ein Gleichheitszeichen (beispielsweise `Frequenz=300`). Die Schlüssel, also die Namen der Felder, werden dabei aus dem Attribut `NAME` des `INPUT`-Tag entnommen.
- ❑ Diese Paare werden anschließend aneinander gehängt, separiert durch ein »&«.

So wird aus den Eingaben der verschiedenen Felder eine zusammenhängende Zeichenkette. Für das Beispiel aus Abbildung 3.9 ist dies

```
Name=Sittich&Frequenz=300&Kernel=2.2.10
```

Wir haben oben ja bereits festgestellt, dass die Art, in der das Serverprogramm die Eingabe bekommt, vom Wert des Attributs `METHOD` abhängt, nämlich als Kommandozeilenoption oder in der Umgebungsvariablen `QUERY_STRING` bei `GET` oder über den Standardeingabekanal bei `POST`. Da die Serveranwendung sich nicht darauf verlassen sollte, dass der Autor der HTML-Datei stets nur ein und dieselbe Methode verwendet, wollen wir Reaktionsmechanismen für beide Arten implementieren.

Datenübergabe

Diese Umgebungsvariable ist übrigens nicht die einzige, die der Server für die CGI-Anwendung bereitstellt. Es sind sogar so viele, dass ich sie hier gar nicht alle aufzählen kann und deshalb nur ein paar wenige erwähne:

Zusätzliche Umgebungsvariablen

- ❑ `CONTENT_LENGTH` enthält die Größe der Anfrage in Bytes.
- ❑ `REMOTE_ADDR` stellt die IP-Adresse des Clientrechners dar, auf dem der Browser läuft und von dem die Anfrage kommt.
- ❑ `SCRIPT_NAME` ist der Name des CGI-Programms, das ausgeführt wird.

Die Serveranwendung

Was soll nun unsere CGI-Anwendung genau tun? Versuchen Sie vor dem Weiterlesen diese Frage selbst zu beantworten.

*Arbeitsweise des
Beispielprogramms*

Das Programm, das ich Ihnen gleich vorstelle, geht in folgenden Schritten vor:

1. Bestimme die Art der Anfrage, das heißt GET oder POST.
2. Zerlege die übergebene Zeichenkette in Paare aus Schlüssel und Wert und speichere diese in einer dynamisch aufgebauten Liste (daher also dieses Thema in diesem Kapitel ...)
3. Erzeuge eine Antwortseite, die die gesendeten Informationen auflistet.

*Rückmeldung beim
Aufrufer*

Als Antwort erwartet der Benutzer nämlich eine Webseite, die ihm den Vollzug der gewünschten Aktion meldet. Deren Übertragung ist sehr einfach: Das CGI-Programm muss nämlich nur den HTML-Code auf die Standardausgabe schreiben, dann sorgt der Server dafür, dass dieser an den Anfragenden geschickt wird. Obwohl das Ergebnis meistens in HTML verfasst sein wird, können Sie auch unformatierten Text senden. Sie müssen nur zu Beginn Ihrer Ausgabe deren Typ spezifizieren, was durch `Content-type` gefolgt vom eigentlichen Typ und einer Leerzeile geschieht. Für HTML, das ich auch im Beispiel verwende, sieht das folgendermaßen aus:

```
cout << "Content-type: text/html" <<endl<<endl;
```

Für einfachen Text (ähnlich wie eine Ausgabe im Shell-Fenster) ersetzen Sie in der vorangegangenen Zeile `html` durch `plain`.

Unsere Anwendung hat damit eine recht kurze `main()`-Funktion:

```
int main()
{
    Liste liste;

    if (analysiereAnfrage(liste) == false)
        antwortFehler();
    else
        antworte(liste);

    return 0;
}
```

Sehen wir uns also die weiteren Funktionen an.

Bestimmung der Anfragemethode

Die Art der Anfrage ist in der Shell-Variablen `REQUEST_METHOD` enthalten. Den Inhalt von Umgebungsvariablen können Sie sehr leicht mit Hilfe der Funktion `getenv()` bestimmen. Diese erwartet den Namen der Variablen in Form eines Zeigers auf ein `char`-Feld und gibt deren Inhalt in derselben Form zurück. In unserem Programm heißt das etwa

*Bestimmung von
Shell-Variablen*

```
bool analysiereAnfrage(Liste& _liste)
{
    // Bestimme die Anforderungsart
    string request_method =
        getenv("REQUEST_METHOD");
```

Pufferung der Eingabe

Anschließend müssen wir für beide Methoden jeweils einen Puffer aufbauen, der die Eingabe aufnimmt.

```
// Puffer fuer uebergebene Daten
char* buffer = 0;
unsigned int len;

// Handle eine POST-Anforderung
if (request_method == "POST")
{
    len = atoi(getenv("CONTENT_LENGTH"));
    buffer = new char[len+1];
    for(unsigned int i=0; i<len; i++)
        cin.get(buffer[i]);
}

// Handle eine GET-Anforderung
if (request_method == "GET")
{
    len = strlen(getenv("QUERY_STRING"));
    buffer = new char[len+1];
    strcpy(buffer, getenv("QUERY_STRING"));
}

// Null-Zeichen zur Terminierung
buffer[len] = 0;

// Kopiere Puffer in String
string eingabe = buffer;
delete[] buffer;
```

Zerlegung der übergebenen Zeichenkette

Nun haben wir die Eingabe in Form einer langen, zusammenhängenden Zeichenkette in der Variablen `eingabe` vor uns. Jetzt stehen wir vor der Aufgabe, diese in Paare aus Schlüsseln und Werten zu trennen. Dazu bedienen wir uns einiger Methoden der Klasse `string`, nämlich `find()`, um einen Teilstring ab einer gegebenen Position zu finden, `length()` zur Bestimmung der Gesamtlänge und `substr(p, n)`, um einen Teilstring ab Position *p* mit *n* Zeichen herauszuziehen. Damit können wir folgendermaßen vorgehen:

```
// Lokale Variablen zur Teilstringsuche
size_t pos = 0;
size_t old_pos = 0;

// Lies Schlüssel/Wert-Paare
while(pos < len)
{
    pos = eingabe.find("&", old_pos);
    // Einziges oder letztes Paar
    if (pos == string::npos)
        pos = eingabe.length();

    // Zerlege das Paar
    string paar = eingabe.substr(old_pos,
                                pos-old_pos);
    size_t eq_pos = paar.find("=");

    string schluessel = paar.substr(0, eq_pos);
    konvertiereLeerzeichen(schluessel);

    string wert = paar.substr(eq_pos+1);
    konvertiereLeerzeichen(wert);

    // Fuege Paar in Liste ein
    _liste.push_back(schluessel, wert);
    old_pos = pos+1;
}
```

Zur Liste und deren Implementierung kommen wir gleich noch.

Erzeugung einer Antwortseite

Eine entsprechende Antwort lässt sich in unserem Fall leicht erzeugen, da wir dort lediglich die übergebenen Werte auflisten wollen. Wir müssen aber darauf achten, eine korrekte und vollständige HTML-Seite auszugeben, da sonst der Browser sie nicht darstellen kann.

```
void antworte(Liste& _liste)
{
    cout << "Content-type: text/html" << endl;
    cout << endl;
    cout << "<HTML>" << endl;
    cout << "<HEAD><TITLE>Eingabe verstanden";
    cout << "</TITLE></HEAD><BODY>" << endl;
    cout << "<H2>Sie hatten folgende Angaben"
        << " gemacht: </H2>" << endl;
    cout << "<UL>" << endl;

    ListElement* tmp;
    for(tmp = _liste.front(); tmp != 0;
        tmp = tmp->naechstes)
    {
        cout << "<LI>" << tmp->schluessel << ": ";
        cout << "<EM>" << tmp->wert
            << "</EM></LI>" << endl;
    }
    cout << "</UL>" << endl;
    cout << "</BODY></HTML>" << endl;
}
```

Das Resultat sehen Sie in Abbildung 3.10. Außer der Schleife dürften Ihnen die Befehle keine Probleme bereiten; zu dieser komme ich gleich noch.



Abbildung 3.10
Als Antwort erzeugen wir eine Liste der übergebenen Parameter.

Die verkettete Liste

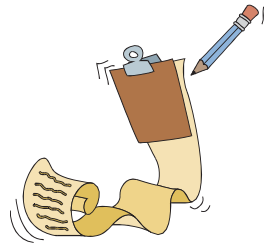
*Dynamische
Datenstrukturen*

Was jetzt noch fehlt, ist die Definition der Listenklasse. Diese führt uns vom Internet zurück zum zentralen Thema dieses Abschnitts, der dynamischen Datenverwaltung. Die Liste soll nämlich nur genauso viel Speicherplatz in Anspruch nehmen, wie sie Elemente enthält. Von diesem Typ gibt es noch einige weitere; bekannt sind etwa

- ❑ die *Schlange* (Englisch *queue*), in der man die Elemente nur in der Reihenfolge wieder bekommt, in der man sie eingetragen hat (auch FIFO-Prinzip genannt, von *first in – first out*),
- ❑ der *Stapel* (*stack*), bei dem man das zuerst wiederbekommt, was als Letztes hinzugefügt wurde (nach dem LIFO-Prinzip, von *last in – first out*),
- ❑ die *Menge* (*set*), die nur paarweise verschiedene Elemente enthalten kann.

Abbildung 3.11

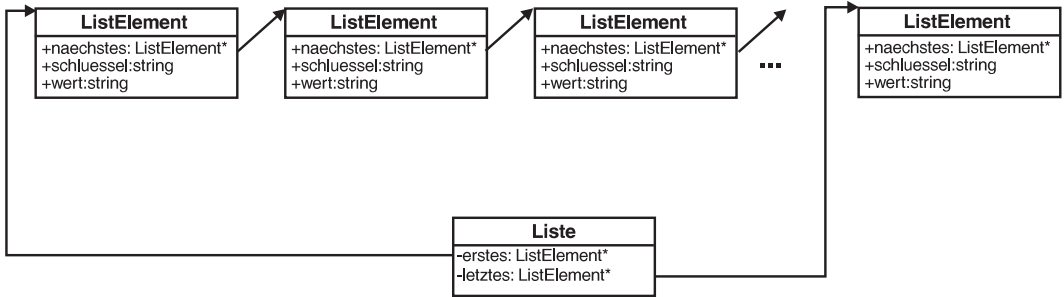
Die Liste ist eine sehr häufig benötigte Datenstruktur.



Leider kann ich an dieser Stelle nicht näher auf das Thema »dynamische Datenstrukturen« eingehen. Ich empfehle Ihnen aber, wenn Sie davon noch wenig gehört haben, sich für Ihre weitere Programmierarbeit mit diesen vertraut zu machen, zum Beispiel anhand des Standardwerks [SEGEWICK 1992]. Auch wenn Sie die meisten Typen nicht selbst implementieren werden, sondern auf die C++-Standardbibliothek zurückgreifen, ist es doch hilfreich, die zugrunde liegenden Prinzipien zu verstehen.

*Die einfach verkettete
Liste*

Eines der einfachsten und (vermutlich deshalb) bekanntesten Beispiele für eine dynamische Datenstruktur ist die *einfach verkettete Liste* (siehe auch Abbildung 3.12). Dabei sind die einzelnen Elemente über einen Zeiger miteinander verbunden. Jedes Element zeigt genau auf ein weiteres. Auf diese Weise können Sie sich vom Anfang bis zum Ende wie an einer Kette durchhangeln. Das Ende ist dadurch gekennzeichnet, dass der Zeiger den Wert 0 hat.

**Abbildung 3.12**

Bei verketteten Listen enthält jedes Element einen Zeiger auf das nächste.

Diese Listenart ist ausreichend, wenn es darum geht, eine vorher nicht bestimmbar Anzahl von Elementen im Speicher festzuhalten, ohne dass große Anforderungen an den Zugriffskomfort zu erfüllen sind. Ihr Nachteil ist nämlich, dass Sie nicht beliebig auf jedes Element zugreifen können, sondern immer vom Anfang zum Ende durchlaufen müssen. Als Verbesserung kann man beispielsweise *doppelt verkettete Listen* erstellen, bei denen jedes Element nicht nur auf seinen Nachfolger, sondern mit einem zweiten Zeiger auch auf seinen Vorgänger weist. Somit ist zumindest der Durchlauf in zwei Richtungen möglich (siehe auch die entsprechende Übungsaufgabe!).

Als Ausgangspunkt für die Definition der Liste dient uns die Elementstruktur. Wir werden später noch sehen, wie man Listen und andere Datenstrukturen generisch, das heißt ohne konkreten Bezug zum Typ des Inhalts, definieren kann. An dieser Stelle aber legen wir eine Struktur für unsere spezielle Aufgabe, die übergebenen Paare aus Schlüsseln und Werten zu speichern, genau fest.

Definition eines Elements

```

struct ListElement
{
    ListElement* naechstes;
    string      schluessel;
    string      wert;

    // Standardkonstruktor
    ListElement() :
        naechstes(0) {}
  
```

```

// Spezieller Konstruktor
ListElement(const string& _schluessel,
            const string& _wert) :
    naechstes(0),
    schluessel(_schluessel),
    wert(_wert)
    {}
};

```

*Datenelemente in der
Listenklasse*

Unsere Klasse `Liste` enthält als private Datenelemente neben der Anzahl der Elemente in der Liste je einen Zeiger auf das erste sowie auf das letzte Element. So können wir sowohl für Durchläufe auf den Anfang zugreifen als auch bequem neue Elemente am Ende einfügen.

```

class Liste
{
private:
    ListElement* erstes;
    ListElement* letztes;
    int anzahl;

```

*Deklaration der
Methoden*

Die Methoden bieten genau die Funktionalität, die wir von der Klasse erwarten.

```

public:
    Liste() :
        erstes(0), letztes(0), anzahl(0) {}

    virtual ~Liste();

    bool empty() const
        { return (anzahl == 0); }

    int size() const
        { return anzahl; }

    void push_back(const string& schluessel,
                  const string& wert);
    void pop_front();

    ListElement* front()
        { return erstes; }
};

```

Dabei ist der Konstruktor ziemlich einfach. Auch die Informationen über die Länge der Liste sind leicht zu implementieren. Die Namen der

Methoden sind übrigens alle von der entsprechenden Klasse der Standardbibliothek übernommen, damit Ihnen und Ihren Programmen der Umstieg später leichter fällt.

Als Nächstes wollen wir uns ansehen, wie man Elemente an das Ende der Liste anfügt.

Anfügen am Ende

```
void Liste::push_back(const string& _schluessel,
    const string& _wert)
{
    ListElement* tmp =
        new ListElement(_schluessel, _wert);

    if (letztes != 0)
        letztes->naechstes = tmp;
    else
        erstes = tmp;

    letztes = tmp;
    anzahl++;
}
```

Zunächst erzeugen wir dynamisch ein neues Objekt vom Typ `ListElement`, welches wir auch gleich über den speziellen Konstruktor mit den übergebenen Werten initialisieren. Wenn die Liste bereits andere Elemente enthält, binden wir das bisherige Ende an das neue Element an. Ansonsten ist dieses Element auch gleichzeitig das erste. Anschließend sorgen wir noch dafür, dass der Zeiger jetzt auf das neue Ende der Liste verweist, und erhöhen den Elementzähler um eins.

Das Gegenstück dazu, nämlich das Entfernen eines Elements vom Anfang der Liste, bietet uns die Methode `pop_front()`.

Entfernen eines Elements vom Anfang

```
void Liste::pop_front()
{
    if (anzahl == 0)
        return;

    ListElement* tmp = erstes;
    erstes = tmp->naechstes;
    if (erstes == 0)
        letztes = 0;

    delete tmp;
    anzahl--;
}
```

Hier merken wir uns die Adresse des bislang ersten Elements, da wir unser Zeigerattribut `erstes` nun auf das zweite setzen. Gibt es kein

solches Element, ist die Liste nunmehr leer. Jetzt können wir das Objekt löschen und die Anzahl vermindern. An diesem Beispiel sehen Sie auch, dass es für den Umgang mit dynamisch verwaltetem Speicher nicht auf die konkrete Variable ankommt, sondern nur auf den Zeiger auf die Speicherstelle. Dieser kann durchaus im Laufe des Programms von verschiedenen Zeigervariablen gehalten werden; es genügt, wenn zum Zeitpunkt der Freigabe in einer davon die Adresse vorhanden ist – und natürlich die Freigabe nur einmal erfolgt.

Destruktor

Mit dieser Methode ausgerüstet gerät unser Destruktor fast schon trivial. Wir müssen nämlich lediglich `pop_front()` so oft aufrufen, bis keine Elemente mehr vorhanden sind.

```
Liste::~~Liste()
{
    while(anzahl != 0)
        pop_front();
}
```

Verwendung der Liste

Jetzt verstehen Sie sicher auch die Verwendung der Liste in der Funktion `antworte()` (auf Seite 236) etwas besser. Wir definieren eine Zeigervariable, die auf das erste Listenelement verweist. Bei jedem Durchlauf geben wir die Inhalte aus und setzen den Zeiger auf das nachfolgende Element. Das können wir so lange machen, bis keines mehr da ist, der Zeiger also auf 0 steht.

```
ListElement* tmp;
for(tmp = _liste.front(); tmp != 0;
    tmp = tmp->naechstes)
{
    cout << "<LI>" << tmp->schluessel << ": ";
    cout << "<EM>" << tmp->wert
        << "</EM></LI>" << endl;
}
```

Versuchen Sie sich die Vorgänge dadurch klar zu machen, dass Sie dieses Codestück als `while`-Schleife umformulieren.

Fazit

Im Grunde ist das Programm, das wir gerade vervollständigt haben, nicht besonders kompliziert – es macht ja auch nicht sehr viel. Und doch hat es uns einige Mühen gekostet, die notwendigen Schritte zu verstehen und bereitzustellen. Wenn Sie auf dieser Grundlage eine größere Serveranwendung erstellen wollen, kann ich Ihnen nur raten, den Aufwand nicht zu unterschätzen. Das, was für den Web-Surfer ganz spielerisch aussieht, bedeutet für den Web-Autor harte Arbeit, sowohl

am Design der Seiten als auch bei der Programmierung der dynamischen Inhalte.

Nichtsdestotrotz sollten Sie Ihre Serveranwendungen mit viel Sorgfalt planen. Denn sie unterscheiden sich in einiger Hinsicht von normalen interaktiven oder Hintergrundanwendungen:

Besonderheiten von Serveranwendungen

- ❑ Je nach Beliebtheit Ihrer Website muss Ihr Server in der Lage sein, viele Anfragen gleichzeitig zu bearbeiten. Testen Sie beispielsweise Ihre Anwendung dadurch, dass Sie sie 100-mal und mehr starten. Wenn Ihr Rechner dabei »in die Knie geht«, ist entweder Ihr Computer zu schwach oder Ihre Anwendung zu aufwändig. Darüber hinaus spielt die gleichzeitige Benutzung von Ressourcen wie Dateien oder Datenbanken eine wichtige Rolle.
- ❑ Jede interaktive Webseite stellt das Starten eines Programms durch einen völlig fremden Client auf Ihrem System dar. Gehen Sie am besten davon aus, dass jeder Client einen Angriff auf Ihr System vorhat, und gestalten Sie Ihre Serveranwendung entsprechend. Vermeiden Sie unkontrollierbare Aufrufe von anderen Programmen über `system()`, da damit im schlimmsten Fall Shells, also unbeschränkte Zugriffe auf Ihren Computer, ermöglicht werden.
- ❑ Obwohl der Benutzer Ihre Anwendung nicht direkt startet, sondern seine Anfrage erst an den Server schickt, sollten Sie dennoch mit aller Art von eintreffendem Unsinn zurechtkommen. Machen Sie Ihr Programm also so robust wie möglich gegenüber Fehleingaben. Es macht nicht nur einen schlechten Eindruck auf die Benutzer, wenn ihre Anfrage gelegentlich mit einem »Server error« beantwortet wird, es kann unter Umständen auch die Stabilität Ihres Servers beeinträchtigen.

Wenn Sie ohnehin schon geplant haben, interaktive Webseiten zu erstellen, werden Sie diese Hinweise vermutlich auch nicht mehr davon abhalten. Andernfalls sollten Sie sich vor Augen halten, wie viele Websites mit weniger Sorgfalt auskommen und auch nicht gleich unter Einbrüchen von Hackern leiden. Dynamische Webseiten machen erst den Unterschied zwischen Internet und Printmedien aus; nutzen Sie diese Möglichkeit, um mit den Betrachtern Ihrer Seiten in Kontakt zu treten. Auch wenn das eigentliche Thema dieses Abschnitts, die dynamische Speicherverwaltung, unter den Details der CGI-Programmierung etwas verschwand, so sollte Ihnen doch anhand der Listenklasse klar geworden sein, welche Handgriffe zu erledigen und welche Stolperfallen zu beachten sind. Doch selbst wenn Sie alle Tipps verinnerlicht haben, werden Sie immer wieder einmal Abstürzen, falschen Speicherzugriffen

und Speicherlecks begegnen. Selbst für erfahrene Programmierer birgt die dynamische Speicherverwaltung in C++ viele Fehlerquellen.

3.3.7 Zusammenfassung

Folgende Aspekte aus diesem Abschnitt sollten Sie im Gedächtnis behalten:

- ❑ Felder (Arrays) können von jedem elementaren Datentyp gebildet werden. Der Zugriff auf die Elemente erfolgt über den `[]`-Operator. Die Größe eines Feldes muss immer durch eine Konstante angegeben werden.
- ❑ Strings (Zeichenketten) werden in C als Felder des Typ `char` dargestellt. Jede Zeichenkette enthält als letztes Zeichen den Wert `0`, der das Ende des Strings anzeigt. In C++ sollte man aber lieber die Klasse `string` der Standardbibliothek verwenden.
- ❑ Ein Zeiger ist eine Variable, die die Speicheradresse einer anderen Variablen enthält. Den Typ eines Zeigers geben Sie durch einen Stern `*` hinter dem Grundtyp an, etwa `int*`. Die Adresse einer Variablen erhalten Sie, indem Sie den Adressoperator `&` davorsetzen. Auf den Inhalt der Speicherstelle, auf die ein Zeiger verweist, können Sie durch einen `*` vor der Zeigervariablen zugreifen (Dereferenzierung genannt). Bei Zeigern auf Objekte können Sie zum Zugriff auf einzelne Elemente auch den Pfeiloperator `->` verwenden, zum Beispiel `ListElement->naechstes`.
- ❑ Auch die Variable eines Feldes fester Größe ist ein Zeiger. Statt über den `[]`-Operator können Sie auch das Feld wie einen Zeiger dereferenzieren.
- ❑ Wenn ein Zeiger noch nicht oder nicht mehr auf ein definiertes Objekt zeigt, sollte er auf `0` gesetzt werden. Dereferenzierungen von Nullzeigern führen zum Programmabsturz.
- ❑ Um den Speicherplatz für ein Objekt oder Feld erst anhand des tatsächlichen Bedarfs zur Laufzeit festlegen zu können, verwendet man dynamisch reservierte Speicherbereiche. Diese müssen vom Programmierer selbst belegt und freigegeben werden.
- ❑ Der Operator `new` reserviert Speicher für das danach angegebene Objekt und liefert einen Zeiger darauf zurück. Der Operator `delete` zum Freigeben braucht einen solchen Zeiger als Argument. Der Speicher ist unabhängig vom aktuellen Gültigkeitsbereich, wie Block oder Methode, belegt, bis er durch `delete` wieder freigegeben wird.
- ❑ Zu jedem `new` muss es ein zugehöriges `delete` geben, sonst entstehen Speicherlöcher. Allerdings darf `delete` nur einmal auf ein

Objekt angewendet werden, sonst stürzt das Programm ab. Die Anwendung von `delete` auf Nullzeiger ist ohne Wirkung.

3.3.8 Übungsaufgaben

1. Beantworten Sie folgende Fragen:

- Warum darf man dem Element `a[10]` keinen Wert zuweisen, wenn das Feld `a` zehn Elemente hat?
- Erklären Sie den Begriff »Nullterminierung«.
- Was ist der Unterschied zwischen einem Feld und einem Zeiger?
- Welche Bedeutung hat der Pfeiloperator `->`?
- Wie ist der Zusammenhang zwischen Zeigern und Referenzen?
- Welche Art von Objekten sollte man auf dem Heap anlegen?
- Was passiert, wenn das Programm dynamisch Hauptspeicher reservieren will, aber keiner mehr verfügbar ist?
- Was ist ein »Speicherloch«?
- Wieso kommt dem Kopierkonstruktor eine besondere Bedeutung zu, wenn das Objekt dynamisch Speicher reserviert hat?
- HTML-Formulare können ihre Daten mit `GET` oder `POST` an den Server schicken. Worin liegt dabei der Unterschied?

2. Welche Probleme finden Sie in folgendem Programm:

```
1: const int LEN=32;
2: typedef struct Foo
3: {
4:     char *ding;
5:     char *dong;
6: };
7:
8: char* f1(void)
9: {
10:     char carray[LEN];
11:     carray[0] = 'a';
12:     return carray;
13: }
14:
15: char* f2(void)
16: {
17:     char *cp;
```

```
18:   int i = LEN;
19:   cp = new char[LEN];
20:
21:   while (i)
22:       cp[i--] = '\\0';
23:
24:   return cp;
25: }
26:
27: char* f3()
28: {
29:     return new char[1024];
30: }
31:
32: int main(void)
33: {
34:     char* cp1;
35:     char* cp2;
36:     char* cp3;
37:     Foo *f;
38:
39:     f = new Foo;
40:     f->ding = new char[128];
41:     f->dong = new char[128];
42:
43:     cp1 = f1();
44:     cp1[0] = 0x01;
45:
46:     cp2 = f2();
47:     cp3 = f3();
48:
49:     cp3 = NULL;
50:     delete f;
51:     return 0;
52: }
```

3. Schreiben Sie ein Programm, das dem Benutzer erlaubt, die Anzahl der Zeilen sowie eine Matrix selbst einzugeben, und das anschließend die Determinante dieser Matrix berechnet. Diese ist definiert als

$$|A| := a_{11}A_{11} - a_{12}A_{12} + a_{13}A_{13} - \dots + a_{1n}A_{1n},$$

wobei die Unterdeterminante A_{kl} aus A durch Streichen der Zeile k und der Spalte l hervorgeht. (Vielleicht kennen Sie aber auch ein

effizienteres Verfahren zur Determinantenberechnung, etwa über Dreieckszerlegung.)

4. Schreiben Sie die Klasse `Liste` aus Abschnitt 3.3.6 in eine doppelt verkettete Liste um. Fügen Sie dazu auch entsprechende Zugriffsmethoden hinzu und definieren Sie einen Kopierkonstruktor.

3.4 Die C-Bibliothek

Von Anfang an war ein Kennzeichen der Programmiersprache C, dass der eigentliche Sprachumfang recht begrenzt ist, dafür allerdings viele Funktionen über eine Standardbibliothek verfügbar sind. In dieser findet der Programmierer sowohl sehr systemnahe Routinen, mit denen er unmittelbaren Zugriff auf die Hardware seines Rechners erhält, als auch allgemeine Ein-/Ausgabe-, String- und mathematische Funktionen, die in fast allen Programmen benötigt werden.

Die C++-Standardbibliothek will die C-Bibliothek nicht ersetzen, genauso wie C++ nicht C vollständig ersetzen will. Die meisten der C-Funktionen sind nach wie vor verfügbar und sind als ein Teil in die C++-Standardbibliothek eingeflossen. Wir haben bisher nur wenig mit den C-Routinen gearbeitet, da sie vielfach Zeiger erfordern und häufig auch durch echte C++-Alternativen ersetzt werden können. In umfangreicheren Programmen werden Sie indessen nicht umhin kommen, einige der C-Funktionen zu verwenden. In diesem Abschnitt will ich Ihnen daher einen knappen Überblick über die wichtigsten Funktionen geben und Sie auf einige typische Anwendungsfälle hinweisen.

3.4.1 Umfang der C-Bibliothek

Zunächst will ich Ihnen ein Gefühl dafür vermitteln, was eigentlich alles zur C-Bibliothek gehört. Später werden wir uns dann einige Details herausgreifen.

Die Header-Dateien

Um eine externe Funktion zu verwenden, müssen Sie üblicherweise eine Header-Datei mit deren Prototypen in Ihren Code einbinden. Die Header der Standardbibliothek folgen der Konvention, dass ihre Dateinamen keine Endungen wie `.h` oder Ähnliches enthalten. Die C-Abkömmlinge sind dabei wieder in eigenen Header-Dateien gekapselt, deren Name derselbe ist wie in Standard-C, allerdings ohne Erweiterung und mit dem vorangestellten Buchstaben `c`. In diesen wird

Header-Dateien