



Datenstrukturen

Verkettete Listen bauen

Das **Programm** aus dem letzten **Beitrag** ist für einen echten **File-Viewer** eindeutig zu dürrftig. Diese **Schwäche** wird jetzt behoben. Bei der Gelegenheit lernen Sie, wie man in **C/C++** mit der dynamischen **Speicherverwaltung** umgeht.

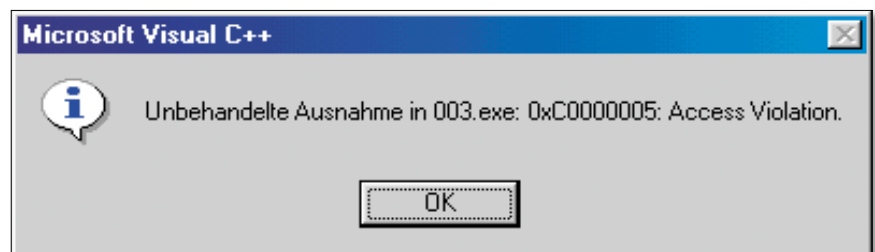
THOMAS WÖLFER

Der erste Versuch war schon ganz nett: Immerhin kann man mit dem Mini-Reader Dateien direkt am Bildschirm anzeigen und das egal wie lang diese Dateien sind. Leider sieht man maximal nur die ersten 100 Zeichen – es fehlt eine Möglichkeit zum Scrollen. Nun könnte man natürlich das Programm so ändern, dass es die komplette Datei jedes Mal liest und dann aber nur die Zeilen anzeigt, die man gerade sehen möchte. Das wäre sicherlich eine Lösung, aber nicht die, die im Folgenden angestrebt wird. Stattdessen soll das im Laufe dieses Beitrags weiterentwickelte Programm Folgendes tun:

- 1.) Die Datei wird komplett gelesen und im Arbeitsspeicher gehalten. Dabei soll mit dem Speicher möglichst sparsam umgegangen werden.
- 2.) Ist die Datei komplett eingelesen, sollen die ersten 24 Zeilen am Bildschirm angezeigt werden. Danach soll das Programm auf einen Tastendruck warten.
- 3.) Wird die Taste "v" gedrückt, wird vorwärts gescrollt. Nach dem ersten Druck sollten die Zeilen 2 bis 25 sichtbar sein. Die Taste "r" scrollt rückwärts, und die Taste "e" beendet das Programm.
- 4.) Vor dem Ende des Programms soll noch der benutzte Speicher freigegeben werden.

Bei dieser Aufgabestellung tauchen bereits einige bisher unbekannte Dinge auf – warum zum Beispiel muss Speicher "freigegeben" werden? – Dazu später

"char" für die Zwischenspeicherung einer Zeile verwendet – warum sollte man nicht ein Array von Arrays anlegen: Dieses zweidimensionale Array hätte dann



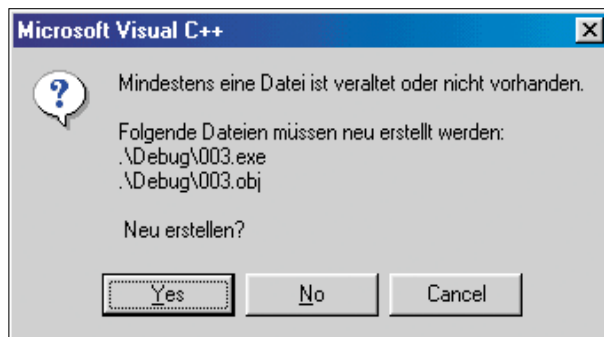
UNBEHANDELTE AUSNAHMEFEHLER: Wenn diese nicht mehr auftreten, ist das Programm hoffentlich stabil und alle Zeiger zeigen dort hin, wo sie hinzeigen sollen. Das sollten Sie auf jeden Fall trotzdem nochmals mit dem Debugger verifizieren.

mehr. Zunächst einmal eine kleine Übersicht über die dynamische Speicherverwaltung im Allgemeinen. Wenn man sich den Vorgang mit den Zeilen überlegt, scheint es eine einfache Lösung für das Problem zu geben: Beim ersten Programm wurde schließlich ein Array aus

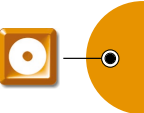
eine Dimension für jede Zeile in der Datei und eine zweite für den Inhalt dieser Zeilen.

Dieser Ansatz birgt aber ein paar Probleme: Zunächst einmal ist es so, dass die Anzahl an Zeilen in einer beliebigen Textdatei nicht bekannt ist. Das ließe sich natürlich lösen, indem man die Datei zweimal lesen würde: Beim ersten Lesevorgang ermittelte man dann diese Anzahl, damit man beim zweiten Lesevorgang die Array-Größe richtig festlegen könnte. Das ist aber nicht sonderlich elegant. Das zweite Problem wäre, dass bei einem solchen Array die Länge der einzelnen Arrays der einzelnen Zeilen alle gleich groß sein müssten – und zwar so groß, wie die längste Zeile in der Datei. Man würde also die Anzahl Zeilen mal der Anzahl Byte in der längsten Zeile Speicher verbrauchen, und das ist inakzeptabel.

Für solche Fälle – also für Fälle, bei denen der konkrete Speicherbedarf erst



WENN SIE DAS PROGRAMM MIT [F5] im Debugger starten, aber noch Dateien zum Übersetzen anliegen, fragt die IDE an, ob das zunächst geschehen soll. Beantworten Sie diese Frage mit nein, debuggen Sie in der Regel die falsche Version Ihres Programms.



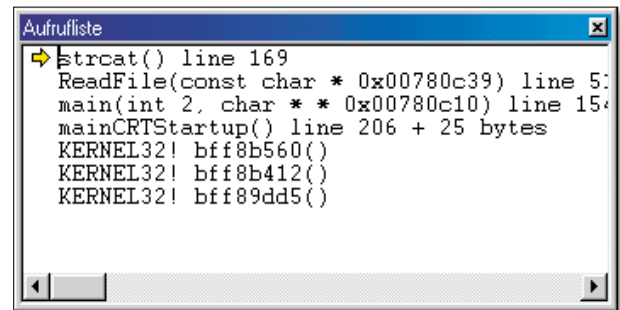
zur Laufzeit eines Programms bekannt wird – ist die dynamische Speicherverwaltung in C/C++ zuständig. Damit kann man zur Laufzeit einen Speicherblock vom Betriebssystem anfordern, benutzen, und dann wieder freigeben. Das geht mit verschiedenen Funktionen, aber in modernen Programmen sollte man dafür immer nur den *new*-Operator einsetzen. Mit dem können Sie sowohl Speicher für eine einzelne Variable anfordern, Sie können aber auch Speicher für ganze Arrays besorgen – wobei die Größe des Arrays erst zur Laufzeit bekannt sein muss.

Der *New* Operator liefert immer einen Zeiger – und zwar entweder einen auf den zur Verfügung gestellten Speicherbereich, oder aber *NULL*: Wird *NULL* geliefert, bedeutet das, dass die angeforderte Menge Speicher vom Betriebssystem nicht mehr zur Verfügung

einer festen Größe verwendet – keine Zeile darf länger als dieser Puffer sein. Das wäre zu vermeiden, würde das Beispielprogramm allerdings deutlich unübersichtlicher machen – daher sei diese Limitierung gestattet. Ansonsten wird im Beispielprogramm nur der Speicher angefordert, der auch wirklich gebraucht wird.

■ Verkettete Liste: die Organisatoren

Für die Arbeit mit dynamischem Speicher gibt es eine ganze Menge Datenstrukturen – aber eine davon passt beim



IN DER AUFRUFLISTE (CALL STACK) finden Sie die Reihenfolge der Funktionsaufrufe, über die das Programm an die aktuelle Stelle gelangt ist.

– vielmehr müssen Sie diesen Datentyp selbst definieren. Das kann man in C/C++ nämlich auch, was unschätzbare Vorteile beim Programmieren bringt. Es gibt wieder verschiedenste Methoden, Datentypen zu konstruieren. Die einfachste davon ist die, ein *struct* zu verwenden. Bei einem *struct* fasst man mehrere Variable zu einer Struktur zusammen und gibt dieser Struktur einen Namen. Danach hat man einen Datentyp, der sich aus mehreren Elementen zusammensetzt, aber ansonsten mehr oder weniger wie einer der eingebauten Datentypen funktioniert. Im Fall von verketteten Listen nennt man diese Elemente meist "Node" (also Knoten), und so sollen diese Elemente auch in diesem Beispiel heißen. Die komplette Definition eines Node hat folgenden Aufbau:

```
struct Node
{
    Node* pPrev;
    Node* pNext;
    char* pLine;
};
```

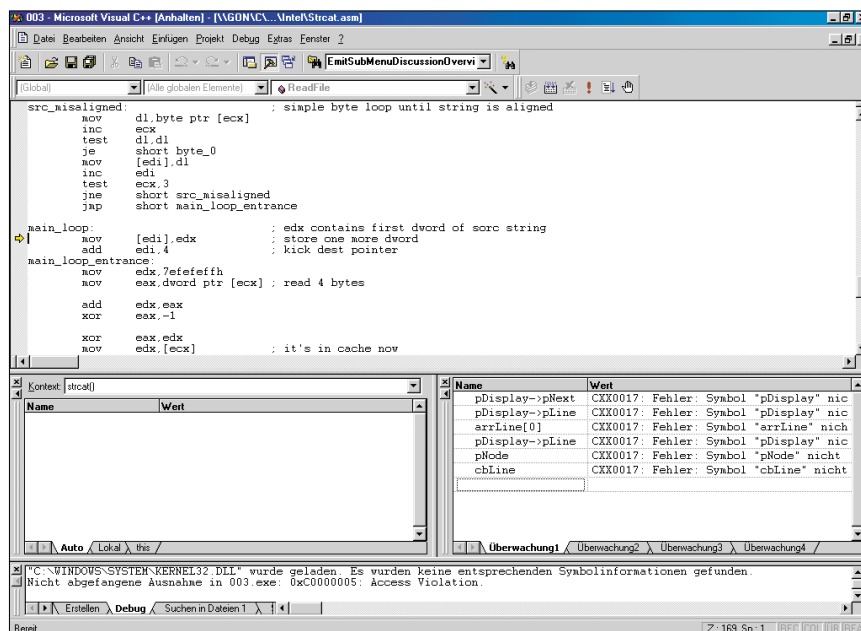
Ein Node enthält also einen Zeiger auf den nächsten Node in der Liste, einen Zeiger auf den vorherigen Node in der Liste sowie einen Zeiger auf einen Speicherbereich, der den Text der Zeile enthält, die zur aktuellen Node gehört.

Und bevor es nun richtig losgeht noch ein bisschen weitere Vorarbeit: Für die Arbeit mit strings – und das ist das, was im Zuge des Programms passiert – benötigen Sie die *string*-Funktionen der Runtime-Bibliothek. Die Prototypen davon befinden sich in der Datei "string.h" – diese muss also zu Beginn des Programms zusätzlich zu den bereits bekannten inkludiert werden.

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

■ Datei einlesen – Knoten allozieren

Der Trick bei der dynamischen Speicherverwaltung ist der, immer nur so viel



WENN SIE DEN QUELLCODE der Runtime-Bibliothek mit installieren oder anderweitig zur Verfügung haben, öffnet der Debugger im Fehlerfall automatisch ein Fenster mit diesem Quellcode, wenn ein Fehler auftritt.

gestellt werden konnte. Mit anderen Worten: Das physische RAM ist vollständig in Benutzung und auch der virtuelle Speicher ist komplett belegt. (Im Beispielprogramm wird dieser Zustand aber weitestgehend ignoriert.). Das anfordern von dynamischem Speicher nennt man übrigens "allozieren" (engl: to allocate).

Doch wird auch bei unserem Beispielprogramm trotz dynamischer Allokierung noch eine Limitierung eingebaut sein: An einer Stelle wird ein Puffer

Beispiel ganz besonders gut, und das sind verkettete Listen. Solche verketteten Listen sind eine Datenstruktur, die sich aus einer beliebig großen Anzahl an Elementen zusammensetzen. Jedes einzelne Element kennt dabei seinen Vorgänger (die vorherige Zeile) und seinen Nachfolger (die nächste Zeile) und hat außerdem ein Element, das die tatsächlichen Daten (also den Text einer Zeile) aufnehmen kann.

Dabei handelt es sich aber nicht um einen in C/C++ eingebauten Datentyp



Name	Wert
pRoot	0x00780eb0
pPrev	0x00000000
pNext	0x00780df0
pPrev	0x00780eb0
pNext	0x00780d80
pPrev	0x00780df0
pNext	0x00780d10
pPrev	0x00780d80
pNext	0x00780c80
pLine	0x00780cc0 "#include <stdafx.h>
pLine	0x00780d50 "
pLine	0x00780dc0 "//
pLine	0x00780e30 "// 003.cpp : Definiert den Einsprungpunkt für die Konsolenanwendung

MIT DEM ÜBERWACHUNGSFENSTER können Sie auch den Elementen der verketteten Liste ganz einfach folgen: Der Debugger baut dazu dann einen kleinen Baum.

Speicher anzufordern, wie Sie auch wirklich brauchen. Genau das passiert nun innerhalb der bereits aus dem Vorgängerprogramm bekannten Schleife (siehe Listing 1).

LISTING 1

```
while( ! feof( pSource))
{
    char arrLine[1024];
    fgets( arrLine, 1024, pSource);
}
```

Zunächst einmal ist die Größe des Arrays auf 1024 Zeichen hochgesetzt worden – damit können nun auch "lange" Zeilen eingelesen werden. Nachdem dieses Array nur einmal benutzt wird und nicht für jede Zeile kopiert wird, ist die Speicherverschwendung an dieser Stelle auch ganz akzeptabel. Die Frage ist nun, wie mit den eingelesenen Daten vorzugehen ist. Ganz einfach: Die werden in einem Node für die verkettete Liste gespeichert. Dieser Node muss zunächst einmal selbst angefordert werden:

```
Node* pNode = new Node;
```

Damit steht nun Speicher für einen einzelnen Node zur Verfügung. Ein einzelner Node hat unter anderem auch einen Zeiger, der die Adresse eines Speicherbereiches für den Text der Zeile aufnehmen kann. Das bedeutet, dass zunächst einmal die Länge dieses Texts ermittelt werden muss, und das geht mit einer der bereits angesprochenen *string*-Funktionen: *strlen()* (STRING LENGTH).

```
int cb = strlen( arrLine);
```

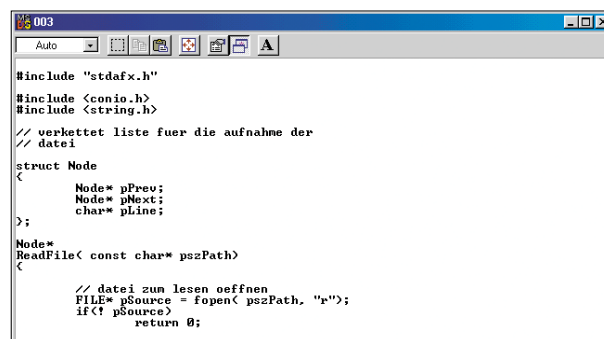
Diese Funktion liefert die Länge eines Strings zurück – allerdings ohne das abschließende NUL-Zeichen (die ab-

schließende Null). Wenn Sie also danach Speicher für den String anfordern wollen, müssen Sie dies berücksichtigen und die Größe des Speicherbereichs entsprechend um 1 erhöhen.

Doch zunächst stellt sich die Frage: Wie erreichen Sie das Element "pLine" aus dem zuvor allozierten Node – schließlich haben Sie gar keine Variable vom Typ Node, sondern kennen nur die Adresse eines Speicherbereiches, in dem die Variable liegt: Das ist schließlich das, was *new()* geliefert hat.

Dafür gibt es eine spezielle Schreibweise in C/C++: Um die einzelnen Members einer Struktur zu erreichen, wenn Sie nur einen Zeiger auf diese Struktur besitzen, benutzen Sie den Operator *->* (Pfeil). Um beispielsweise an das pLine-Member über den pNode-Zeiger zu gelangen, verwenden Sie das folgende Statement:

```
pLine->pNode
Man sagt dazu "pLine zeigt auf pNode".
```



DIE NEUE PROGRAMMVERSION ist schon deutlich besser als die alte: Nun kann man auch lange Dateien anzeigen und scrollen.

Nachdem das geklärt ist, können Sie nun den Speicher für die Zeile anfordern und sich den gelieferten Zeiger entsprechend merken:

```
pLine->pNode =
new char[ cb + 1];
```

Sie fordern also Speicher für "cb + 1" Elemente vom Typ "char" an und merken sich die gelieferte Adresse in "pLine->pNode". Nun können Sie den Speicher verwenden – und das tun Sie, indem Sie die Zeile aus arrLine in den soeben angeforderten Speicherblock der passenden Größe kopieren. Dazu gibt es eine weitere spezielle Funktion namens *strcpy* (STRING CoPY):

```
strcpy( pNode->pLine, arrLine);
```

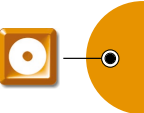
Soweit, so gut: Damit können Sie alle Zeilen lesen und in Speicherblöcken der passenden Größe unterbringen: Jeder Zeiger "pLine" der einzelnen "pNodes" zeigt dann auf einen Block einer genau passenden Größe.

Ganz schlecht: verlorene Zeiger

Das ist nun schon ganz nett – nur wenn Sie die Schleife verlassen, geht's nicht so recht weiter: Sie haben zwar die komplette Datei irgendwo im RAM abgelegt – nur wo? Da tut sich eine weitere wichtige Eigenschaft beim Programmieren mit C/C++ auf: Sie dürfen Zeiger, die Sie einmal angefordert haben, nicht "vergessen". Das bedeutet, Sie müssen sicherstellen, dass Sie immer eine Variable haben, in der die Adresse des Speicherbereichs auch abgelegt ist – ansonsten ist der Speicher verloren, und Sie haben ein "Speicherleck".

Im Falle der verketteten Liste ist das aber ganz einfach – alles, was Sie tun müssen, ist die Verkettung herstellen und sich dann einen Zeiger auf ein einziges Element in der Kette merken: Das reicht völlig aus, um alle Elemente der Liste erreichen zu können. Diese Verkettung stellen Sie am besten direkt in der Schleife her, in der Sie die Nodes für die Liste auch anfordern.

Wie bereits erwähnt, hat ein Node neben dem Zeiger für die Daten auch je einen Zeiger auf das nächste bzw. auf das vorherige Element. Dabei ist es noch wichtig zu wissen,



wann es kein "nächstes" bzw. "vorheriges" Element mehr gibt. Kein "vorheriges" Element gibt es dabei logischerweise beim ersten Element in der Liste – und das letzte Element hat kein folgendes. Diese beiden Sonderfälle markieren Sie einfach dadurch, indem Sie die entsprechenden Members dieser beiden speziellen Nodes auf 0 setzen.

Außerdem ist es hilfreich, wenn Sie sich im Zuge der Schleife zwei weitere Elemente merken: Das "aktuelle" Element, und das "vorherige" Element. Auch das tun Sie natürlich mit Zeigern, wobei der Zeiger auf "vorherige" Elemente so lange auf 0 zeigt, bis es auch tatsächlich ein vorheriges Element gibt.

Mit diesen Informationen können Sie dann Ihre Verkettung zusammenbauen (siehe Listing 2).

Die Verkettung entsteht aus mehreren Arbeitsschritten, die nacheinander erledigt werden müssen. Zunächst setzen Sie den Zeiger `pCurrent->pLast` – also den "Vorgänger"-Zeiger auf `pLast`. `pLast` seinerseits steht im ersten Schleifendurchlauf auf 0 und wird danach auf `pCurrent` gesetzt. Das bedeutet, dass die erste Node eben keinen (0) Vorgänger hat, während alle späteren einen Zeiger auf das zuvor in der Liste eingefügte Element haben.

Dann überprüfen Sie, ob der `pLast`-Zeiger bereits gesetzt ist. Das ist nur im ersten Durchlauf nicht der Fall, in den weiteren Durchläufen enthält `pLast` die Adresse des Node, der im jeweils vorherigen Schleifendurchlauf erzeugt wurde. Das ist das Vorgänger-Element, und von diesem müssen Sie den "Nachfolger"-Zeiger setzen: `pLast->pNext = pCurrent`.

Schließlich müssen Sie noch sicherstellen, dass der Nachfolge-Zeiger des aktuellen Elements auf 0 gesetzt wird: Gibt es einen weiteren Schleifendurchlauf und damit ein Folgeelement, so wird der Zeiger in diesem folgenden Durchlauf richtig gesetzt. Was Sie hier im Beispielcode nicht sehen (was aber auf der CD enthalten ist), ist die Tatsache, dass

Sie sich außerdem noch einen Zeiger auf eines der Nodes in der Liste merken müssen: Welches ist egal, aber das aller-

```

003.cpp
Node*
ReadFile( const char* pszPath)
{
    // datei zum lesen oeffnen
    FILE* pSource = fopen( pszPath, "r");
    if(! pSource)
        return 0;

    Node* pRoot = 0;
    Node* pCurrent = 0;
    Node* pLast = 0;

    while(! feof( pSource))
    {
        // das ist eigentlich nicht sauber, soll aber
        // so mal reichen...
        char arrLine[ 1025];
        fgets( arrLine, 1024, pSource);

        // speicher fuer die echte laenger der zeile
        // sowie fuer einen neuen knoten besorgen
        // todo: eigentlich muesste man hier testen,
        // ob auch speicher vorhanden ist...
        Node* pNode = new Node;
    }
}

```

DER QUELLCODE-EDITOR VON VC++ verwendet eine farbliche Markierung für Elemente wie Kommentare und C++-Schlüsselwörter.

erste Element bietet sich an – genau das passiert auch im Beispielprogramm von der CD.

■ Daten anzeigen

Der erste Teil ist getan: Die Textdatei wird nun in einer verketteten Liste gespeichert. Dass das auch wirklich funktioniert, sollten Sie aber auf jeden Fall im Debugger ausprobieren: Achten Sie dabei darauf, wie die einzelnen Zeiger gesetzt werden – und wo sie hinzeigen. Stürzt die Sache ab, haben Sie mit recht großer Sicherheit mit einem Zeiger an eine Adresse gezielt, an die Sie ihn besser nicht hätten zeigen lassen sollen. Wie auch immer, das Anzeigen der Daten kann nur mit Hilfe des gemerkten "ersten" Elements erfolgen. Dazu wird eine Funktion Namens *DisplayFile()* implementiert. Diese Funktion erhält einen Parameter, und das ist der Zeiger auf eben dieses erste Element in der verketteten Liste:

```
void DisplayFile( Node* pNode);
```

Die Iteration über die verkettete Liste ist verhältnismäßig einfach.

Auch die Zeigerbehandlung wird Ihnen ganz sicher einfach vorkommen, wenn Sie erst ein bisschen Übung darin haben. Alles, was Sie immer wieder tun müssen, ist sich klar zu machen, ob Sie es mit einer Zeigervariablen oder einer normalen Variablen zu tun haben. Und das geht mit dem Debugger wirklich ganz einfach.

Zurück zum Anfang: Die Iteration über die verkettete Liste ist recht einfach, denn schließlich kennen Sie für jeden Node in der Liste das nächste und das vorhergehende Element. Zeigt der Zeiger, der eigentlich auf vorhergehendes Element zeigen soll, auf 0, begutachten Sie gerade das erste Element. Zeigt der Zeiger, der eigentlich auf das nächste Element zeigen sollte auf 0, sind sie bei der letzten Node angekommen.

Die Grundidee bei der Anzeige der Texte ist nun die, dass Sie bei einem Node beginnen, und dann einfach diesen und die 23 nächsten am Bildschirm anzeigen – natürlich nur dann, wenn es auch noch 23 nächste gibt. Um die Anzeige der Texte zu vereinfachen, definieren Sie dazu eine Funktion namens *DisplayNode()*, die sich einzig

LISTING 2

```

Node* pPrev = 0; // vorheriges Element
Node* pCurrent = 0; // aktuelles Element
while( ! feof( pSource))
{

    char arrLine[1024];
    fgets( arrLine, 1024, pSource);

    // Neuer Knoten:
    Node* pNode = new Node;

    // laenge der Zeile
    int cb = strlen( arrLine);

    // speicher fuer die Kopie
    pNode->pLine = new char[ cb + 1];
    strcpy( pNode->pLine, arrLine);

    // verkettung:
    pCurrent = pNode;
    pCurrent->pPrev = pLast;
    if( pLast) pLast->pNext = pCurrent;
    pCurrent->pNext = 0;
    pLast = pCurrent;
}

```




um die Anzeige eines einzelnen Node kümmert:

```
void DisplayNode( Node* pNode)
{
    printf( pNode->pLine);
}
```

Die Schleife zur Anzeige der 23 Zeilen kann dann im Kern wie in Listing 3 aussehen.

Sie iterieren also zunächst über die Anzahl der anzuzeigenden Zeilen. Dann überprüfen Sie, ob der pDisplay-Zeiger gesetzt ist, das heißt, nicht auf 0 zeigt. Ist das der Fall, zeigen Sie den entsprechenden Node an. Dann setzen Sie pDisplay auf seinen eigenen Nachfolger. Gibt es noch einen – ist der Zeiger also nicht 0 – so wird dieser Nachfolger im nächsten Schleifendurchlauf angezeigt. Ist der Zeiger aber 0, so wird auch pDisplay gleich 0.

Im nächsten Schleifendurchlauf wird also nichts mehr angezeigt werden. (Im Beispielprogramm auf der Heft-CD wird das etwas umständlicher in mehreren Schritten getan – das ist deshalb so, damit Sie mit dem Beispielquellcode mehr im Debugger herumprobieren können.) So kann also eine Bildschirmseite angezeigt werden. Fehlt noch die Funktionalität zum Scrollen und Beenden des Programms. Das erledigen Sie einfach mit einer weiteren Schleife, die Sie um die gerade programmierte herumprogrammieren (siehe Listing 4).

Zunächst initialisieren Sie eine Variable mit dem Wert "x", der aber nicht weiter von Interesse ist. Dann starten Sie eine Schleife, die erst dann abbricht, wenn die Variable "cSelect" den Wert "e" annimmt. In dieser Variablen speichern Sie das Zeichen, das die Funktion *getch()* liefert. Ist dieses Zeichen ein "v", so scrollen Sie vorwärts. Bei einem "r" scrollen Sie rückwärts. Das Scrollen selbst erledigen Sie einfach dadurch, dass Sie den Zeiger auf die erste anzuzeigende Zeile auf seinen Nachfolger bzw. seinen Vorgänger setzen – natürlich nur dann, wenn es auch noch einen Nachfolger bzw. Vorgänger gibt. Fertig ist die ganze Scrollerei – das Programm taugt nun schon deutlich mehr als zuvor. Allerdings produziert das Programm ein großes Speicherleck, denn der dynamisch angeforderte Speicher wird zwar in der verketteten Liste prima verwaltet, aber niemals freigegeben.

Das passiert aber im Beispielprogramm auf der Heft-CD: Dort finden

Sie die passende Funktion unter dem Namen *FreeMemory()*. Sie sollten aber zunächst selbst versuchen, den Speicher am Ende des Programmes freizugeben: Dazu benötigen Sie den

operator *delete* – das Gegenstück zu *new*.

Im nächsten Teil des Sonderheftes gibt's noch mehr Zeiger und zwar solche auf Funktionen. UR

LISTING 3

```
Node* pDisplay = pFirstLine // hier auf die erste zeigen
↳ lassen
for( int i=0; i<24; i++)
{
    if( pDisplay)
    {
        DisplayLine( pDisplay);
        pDisplay = pDisplay->pNext;
    }
}
```

LISTING 4

```
char cSelect = 'x';
Node* pFirstLine = pFirstNode;
while( cSelect != 'e')
{
    // Hier steht die andere Schleife

    cSelect = getch();

    if( cSelect == 'v')
    {
        if( pFirstLine->pNext)
            pFirstLine = pFirstLine->pNext;
    }
    if( cSelect == 'r')
    {
        if( pFirstLine->pPrev)
            pFirstLine = pFirstLine->pPrev;
    }
}
```

KOMMENTARE – QUELLCODE ERKLÄREN

Nicht alles, was Sie in ihrem Quellcode unterbringen, muss auch notwendigerweise an den Compiler weitergegeben werden: So ist es beispielsweise immer wünschenswert, wenn der Quellcode mit leicht verständlichen Anmerkungen und Kommentaren versehen wird. Damit kann man zum Beispiel auf Quellangaben verweisen, oder man erläutert bestimmte Algorithmen.

Um Kommentare im Quellcode einzubetten und so vor dem Compiler und dessen Syntax-Check zu verbergen, gibt es zwei Möglichkeiten: einzeilige Kommentare und mehrzeilige Kommentare.

Ein einzeiliger Kommentar kann an einer beliebigen Stelle innerhalb einer Quellcodezeile auftauchen und wird durch einen doppelten Slash (//) eingeleitet.

Alles, was nach diesem doppelten Slash in der Zeile steht – und zwar bis zum Zeilenende, wird vom Compiler ignoriert. Die andere Alternative sind mehrzeilige Kommentare.

Diese beginnen mit einem /* und enden mit dem Zeichen-Paar */. Alles, was zwischen diesen beiden Paaren steht, wird vom Compiler ignoriert. Damit können Sie zum Beispiel auch testweise Quellcode auskommentieren – und das auch auf einer einzelnen Zeile. Dazu ein kleines Beispiel:

```
printf("hallo"); /* printf("C++");
↳ */ printf("Welt");
```

Das mittlere *printf()* befindet sich hier in einem Kommentarbereich – das übersetzte Programm würde nur den Text "halloWelt" ausgeben.