CHIP SPECIAL
Programmierpraxis

84029/93003
34,– DM

Sämtliche
Sources
auf Diskette
im Heft

C

Programmier-Praxis mit C

Ausgabe
11

- Visual C++: Was der neue Compiler bringt
- Hashing: Das schnellste Suchverfahren
- Transputer-Programmierung in C
- Grundalgorithmen zu Graphen
- Alles über Pointer: Listen und Bäume

Markus Mück

Bessere Datenorganisation

Der Kampf um den besten Platz unter den Datenbankprogrammen hat längst begonnen. Wohlklingende Namen konkurrieren untereinander um die besten Benchmarks, die größte, theoretisch verwaltbare Menge an Daten, die mannigfaltigsten Funktionen und so weiter. Verleitet von diesen Versprechungen, haben viele Programmierer Abstand von bewährten Programmiersprachen wie Pacal und C genommen.

Alle zu diesem Beitrag gehörenden Files finden Sie im SubDir MÜCK auf beiliegender Diskette

Stattdessen schlagen sie sich mit teilweise absolut unausgereiften, völlig inkompatiblen Datenbanksprachen herum, welche von modernen Entwicklungsumgebungen, integrierten, wenn überhaupt vorhandenen Debuggern und Toolboxen verschiedenster Art bisher wenig gehört zu haben scheinen.

In vielen Fällen hätte diese Fahnenflucht jedoch überhaupt nicht stattfinden müssen. In der Praxis geht es doch nur selten um die Verwaltung von vier Billionen Datensätzen, komplizierten Indizierungsalgorithmen und ähnlichem. Meist verlangt der Kunde, beispielsweise innerhalb einer speziellen Textverarbeitung, nach einer einfachen Verwaltung von mehreren hundert, in Einzelfällen einigen tausend Kunden. Selbst ein wenig langsamere Algorithmen für die Datenverwaltung wird er tolerieren, wurde doch das restliche Programm durch den Einsatz bewährter Toolboxen profigerecht aufbereitet. Gerade diese stehen für Standardprogrammiersprachen wie C in fast unüberschaubarer Anzahl zur Verfügung. Aus eben diesem Grund werden die folgenden Zeilen den interessierten Leser in die Welt der Organisation von Datensätzen im Speicher über einfache und doppelt verkettete Listen einführen. Diese Methoden lassen eine einfach verwaltbare, dynamische Organisation zu, welche sich leicht auf spezielle Probleme zuschneiden läßt.

Ein Beispiel: Wir haben vor, drei Zahlen im Speicher der Größe nach geordnet zu verwalten. Ein Prinzipbild (Abb. 1) verdeutlicht die Lösung über eine einfach verkettete Liste.

Das Prinzip ist schnell verstanden; eine festgelegte Pointer-Variable zeigt auf das erste Glied der Kette (in unserem Fall das Feld mit der Zahl 1). Dieses Feld enthält nun neben der eigentlichen Daten (die Zahl 1) einen Zeiger auf das folgende Feld (hier das Glied mit Zahl 2). In C könnte der Programmierer das Feld wie folgt gestalten:

```
typedef struct DatenSTRUCT
{
    unsigned int  Zahl;
    unsigned int  *Next;
} Daten;

Daten* Anfang;

Anfang          = (Daten*) malloc ( sizeof (Daten) );
Anfang -> Zahl = 1;
Anfang -> Next = (unsigned int*) malloc( sizeof(Daten) );
( (Daten*) Anfang -> Next ) -> Zahl = 2;
( (Daten*) ( (Daten*) Anfang -> Next ) ) -> Next = (unsigned int*)
            malloc( sizeof(Daten) );

( (Daten*) ( (Daten*) ( (Daten*) Anfang )
 -> Next ) -> Next )->Zahl=5;
```

Ungewohnt gibt sich der in unserer Struktur vorgestellte Zeiger "unsigned int *Next". Eigentlich müßte er auf die Struktur gerichtet sein, in welcher er selbst definiert wird, was aus syntaktischen Gründen nicht möglich ist. Aus diesem Grund definieren wir einfach einen anderen Zeiger und konvertieren beim erneuten Anfordern von Speicher für ein neues Glied in der Kette den Zeiger auf die "Daten"-Struktur in einen Zeiger, welcher auf eine "unsigned int"-Variable zeigt. Ein ungewohnte, aber problemlos realisierbare Möglichkeit.

Wie zuvor angesprochen, haben wir vor, die einzelnen Zahlen aufeinanderfolgend abzulegen, d.h. die Zelle von Zahl 1 soll auf Zahl 2, diese wiederum auf die nächst höhere Zahl und so weiter zeigen. Wie aber gestaltet sich ein Einfügen neuer Einträge in unsere Kette? Abbildung 2 zeigt näheres.

Die schematische Darstellung zeigt, daß es nicht darauf ankommt, wie die einzelnen Zellen im speicher angeordnet sind. Wichtig ist nur, daß eine Zelle auf die nach unseren Vorstellungen nächste zeigt. Der Programmierer bemerkt von der verstrickten, völlig unübersichtlich angeordneten Lage einzelner Zellen nichts. Für ihn zeigt eine auf die andere; eine perfekte Ordnung wird vorgetäuscht.

Dementsprechend einfach gestaltet sich das Einfügen selbst: Das Programm hat festzustellen, an welcher Stelle die neue Zelle Platz zu finden hat. Der Zeiger auf die darauffolgende wird in unsere neue Zelle kopiert und die vorhergehende mit einem Zeiger auf die neue überschrieben. Ein praktisches Beispiel:

```
typedef struct DatenSTRUCT
{
    unsigned int  Zahl;
    unsigned int  *Next;
} Daten;

Daten*        Anfang;
unsigned int* Help;

Anfang        = (Daten*) malloc ( sizeof (Daten) );
Anfang -> Zahl = 1;
```

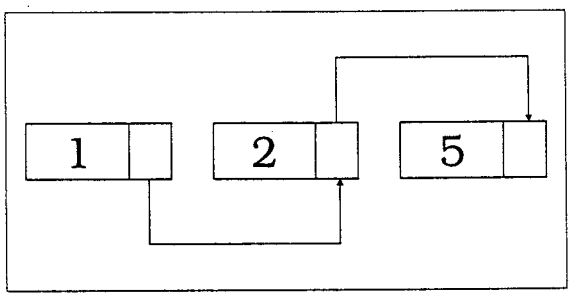

Bild 1: Einfach verkettete Liste

```
/* Erstes Glied enthÊlt Zahl '1' */

( (Daten*) Anfang ) -> Next =
                    (unsigned int*) malloc( sizeof(Daten) );

/* Zeiger im ersten Glied zeigt auf nachfolgendes Glied */

( (Daten*) Anfang -> Next ) -> Zahl = 3;

/* Zweites Glied enthÊlt Zahl '3'                          */
/*********************************************************/
/* Im folgenden wird zwischen das 1. und 2. Glied ein weiteres*/
/* eingefÅgt:                                              */

Help = ( (Daten*) Anfang ) -> Next;
/* Zeiger auf 2. Glied in Zwischenspeicher */

( (Daten*) Anfang ) -> Next =
                    (unsigned int*) malloc( sizeof(Daten) );
/* Zeiger im ersten Glied zeigt auf neues, jetzt 2. Glied */

( (Daten*) Anfang -> Next ) -> Zahl = 2;
/* Neues 2. Glied mit Wert '2' belegen */

( (Daten*) ( (Daten*) Anfang -> Next ) ) -> Next = Help;
/* Neues 2. Glied zeigt auf neues 3. Glied (ehemals 2. Glied) */
```

Jeder aufmerksame Leser dürfte sich an dieser Stelle in der glücklichen Lage befinden, Daten aller Art in einfach verketteten Listen ablegen und damit im Speicher verwalten zu können. Einfache Suchalgorithmen, schon

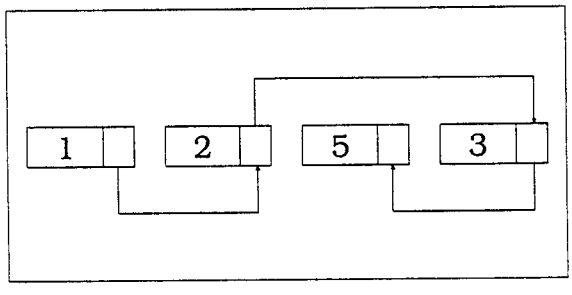

Bild 2: Ein neues Glied in der einfach verketteten Liste

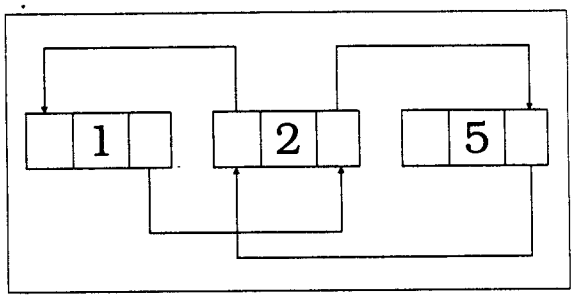

Bild 3: Doppelt verkettete Liste

angefangen bei schnörkellosen Vergleichsroutinen, leisten hierbei gute Dienste, um spezielle Daten aufzufinden. Weitere Verwendungsmöglichkeiten, beispielsweise eine Adressenliste, bedürfen wohl keiner näheren Erläuterung. Hier ist der Ideenreichtum des einzelnen gefordert, aus den unendlichen Möglichkeiten dieser Technik das Vielversprechendste herauszuholen. Aus diesem Grund werden die nächsten Zeilen eine Weiterentwicklung dieser Vorgehensweise vorstellen und nicht auf Deatilerläuterungen des eben Vorgestellten eingehen. Ein Glied einer einfach verketteten Liste besteht im Primitivstfall aus zwei Einheiten: Einmal speichert es bestimmte Daten, zum anderen birgt es einen Zeiger auf das nächste Glied. Die Weiterentwicklung davon gestaltet sich recht einfach. Man denke sich zu jedem Glied einfach einen Zeiger auf das vorhergehende hinzu; schon ist der Schritt zur doppelt verketteten Liste getan. Abbildung 3 verdeutlicht die Problematik schematisch.

Ein typisches Problem, bei welchem die doppelt verkettete Liste Anwendung findet, ist die Textverarbeitung. Einfach dargestellt, findet jede Textzeile in je einem Glied der Kette Platz. Nimmt der User nun Veränderungen an dieser Zeile vor, hat das Programm lediglich einen dynamisch angelegten Text zu ändern. Funktionen wie "realloc" erleichtern die Sache dabei ungemein. Beim Löschen eines Zeichens aus der aktuellen Zeile, wird das Programm den besagten String umberechnet und die Speicherplatzanforderung um 1 Byte erniedrigen. Das Problem ist gelöst.

Erdreistet sich der Anwender, via Cursortasten seine aktuelle Position im Text zu wechseln, verlangt auch dieser Schritt keine Verrenkungen mehr vom Programm ab. Ein einfaches Auslesen der Adresse der nächsten oder vorhergenden Zeile löst komplizierte Berechnungsalgorythmen ab. Demnach gilt es, die oben angewandte Struktur wie folgt zu erweitern:

Aus

```
typedef struct DatenSTRUCT
{
    unsigned int  Zahl;   /* Daten */
    unsigned int  *Next;  /* Zeiger auf nÊchstes Glied */
} Daten;
```

wird

```
typedef struct DatenSTRUCT2
{
    unsigned int  Zahl;   /* Daten */
    unsigned int  *Next;  /* Zeiger auf NÊchstes Glied */
    unsigned int  *Prev;  /* Zeiger auf vorhergehendes Glied */
} DoppelDaten;
```

Die programmtechnische Realisierung entspricht praktisch der anfangs aufgeführten und soll deshalb nicht näher ausgeführt werden. Nun liegt es an ideenreichen Lesern, diese Technik optimal einzusetzen.