CHIP PROGRAMMIER-PRAXIS 84029/92005
34,– DM

C

Sämtliche Sources auf Diskette im Heft

Programmier-Praxis mit C

- Unix-Utilities auf MS-DOS
- Unix-Shell selbst gemacht
- Superschnelles Fenster-System in C
- Assembler in C-Bibliotheken einbinden
- Grafik im Textmodus
- Projekt: Universeller Dialog-Generator, Dialog-Modelle und Schnittstellen-Konzepte

Ausgabe 10

Markus Mück

Allerlei Zeigerei

Die Erfinder von "C" schufen diese Sprache, um über ein universell einsetzbares Werkzeug verfügen zu können. Einmal erstellte Programme müssen nach dem Compilieren mit einem entsprechenden C-Compiler auf jedem nur erdenklichen Rechner lauffähig sein. UNIX-Profis sollten genauso in den Genuß dieser Sprache, wie DOS- oder gar AMIGA-User kommen. Die Idee scheint recht plausibel, wäre es damit doch möglich, ein einmalig geschriebenes Programm auf andere Rechensysteme mit minimalem Arbeitsaufwand zu übertragen.

Mit Zeiger lassen sich universelle Speichermodelle realisieren

Ein besonderes Problem stellt bei diesem Vorhaben die Speicherverwaltung dar. Ein gewöhnlicher PC arbeitet beispielsweise mit einer Segment-Offset-Adressierung, einer Zweiteilung der Speicheradresse also. Diese Art und Weise steht allerdings nicht allein im Dschungel der verschiedensten Computersystemen da. Andere sind beispielsweise in der Lage, den gesamten Speicher ohne eine Aufteilung der Adresse anzusprechen. Problematisch gestaltet sich also das systemunabhängige Ansprechen von Speicher. Die Schöpfer von "C" sind dabei auf eine recht einfache Lösung gekommen. Im Quellcode selbst unterscheidet sich eine Speicheransteuerungsvariable (Pointer oder deutsch Zeiger) kaum von anderen Artgenossen, lediglich ein Sternchen ("*") charakterisiert sie als solche. Der Vorteil liegt auf der Hand: Ein Programmierer interessiert sich nicht mehr für die systeminterne Speicherverwaltung. Sollte ein System eine Zweiteilung der Adressen für notwendig halten, so wird sich der Compiler darum kümmern, nicht der Programmierer. Selbst bei exotischen Adressierungsmethoden ändert sich keine Zeile im Quellcode, Zeiger bleibt Zeiger. An dieser Stelle soll ein kleines Beispiel helfen, die Funktionsweise von Zeigern zu verstehen:

Der Unterschied zwischen den beiden Variablen "x" und "y" liegt in der systeminternen Verwaltung. "x" repräsentiert eine 1-Byte-Zahl, welche in irgendeiner Speicherzelle abgelegt wird. Verändert unser Programm demnach den Wert der Variable "x", wird an derjenigen Speicherzelle, welche für ihren Wert reserviert wurde, der neue eingetragen. "y" haben wir als Zeiger ("int *y") definiert. Diese Variable steht demnach nicht anstelle einer Speicherzelle, welche einen 1-Byte-Wert beinhaltet. "y" kann genaugenommen mit keinem Zahlenwert belegt werden, der für diese Variable reservierte Speicher enthält eine Speicheradresse. Eben diese enthält die Adresse anderer "int"-Variblen, da "y" als Zeiger auf Variablen dieses Typs definiert wurde ("int *y"). Die Anweisung

```
y = &x;
```

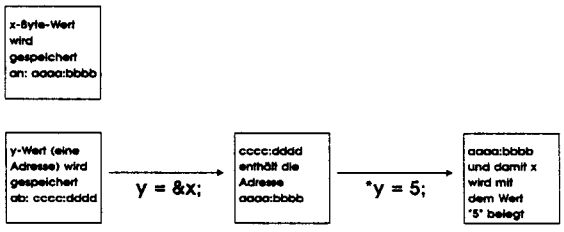
bewirkt, daß der für "y" reservierte Speicherplatz diejenige Adresse enthält, an welcher der 1-Byte-Wert für "x" gespeichert wird (der Adressoperator "&" verrät die Adresse, an welcher der Wert einer bestimmten Variable abgelegt ist). Die Anweisung

```
*y = 5;
```

bewirkt, daß der Computer die Adresse einliest, welche "y" speichert und diese Adresse

mit dem Wert "5" belegt. Da in unserem Fall "y" die Adresse von "x" belegt, oder besser gesagt, "y" auf "x" zeigt, belegt diese Anweisung die für "x" belegte Speicherzelle mit dem Wert 5. Das Resultat: "x" hat nach dem Aufruf dieser Zeile den Wert "5".

Eine graphische Darstellung:




```
#include <stdio.h>

main()
{
int x, *y;

y  = &x;
*y = 5;

  printf("%d", x);
}
```

Nach dem Start gibt das Programm die Zahl "5" auf dem Bildschirm aus.

Abb. 1: Demoprogramm

In der schier unüberschaubaren Vielzahl von Variablentypen setzen sich besonders Strings von der Masse ab. Dieser Typ verfügt über einige Eigenarten, die vor allem Anfänger an den Rand der Verzweiflung bringen. Eine eingehende Erläuterung kann hierbei jedoch schnell Abhilfe schaffen. Auch an dieser Stelle wird uns ein kleines Beispielsprogramm mit alltäglichen Situationen vertraut machen: Die Anweisung

```
char text[10];
```

reserviert 11 Speicherzellen (Speicherzelle 0 bis einschließlich 10) für je ein ASCII-Zeichen. Die Anweisung

```
text[0] = 'H';
```

belegt die erste der elf reservierten Speicherzellen mit dem Zeichen "H". Die an dieser Stelle verwendeten Anführungszeichen (') weisen darauf hin, daß es sich um ein einzelnes Zeichen handelt, kann eine einzelne Speicherzelle doch logischerweise nicht mit einem ganzen String belegt werden. Für letzteres müßten andere Anführungszeichen eingesetzt werden (""). String-Variablen haben außer der eben vorgestellten noch eine andere Eigenschaft. Der Variablenname allein, hier "text", enthält einen Zeiger auf das erste Zeichen des Strings, d.h. der Zeichenkette. Eine Anweisung wie

```
*text = 'G';
```

würde also der Anweisung

```
text[0] = 'G';
```

entsprechen und die erste String-Speicherzelle mit dem Zeichen "G" belegen. Unser Programm nutzt die Möglichkeiten von "C" noch weitergehend aus:

```
*(text + 1) = 'i';
```

Diese Anweisung belegt das zweite Zeichen des Strings (Zeiger auf die Adresse des ersten Zeichens plus eins) mit einem 'i' und entspräche der Anweisung

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

Nun soll eine Standard-Routine der C-Library helfen, einen kleinen String hinter die ersten beiden Buchstaben des Strings zu kopieren. Ihr Syntax lautet:

```
char *strcpy( char *ziel, const char *quelle );
```

Es fällt auf, daß diese Funktion die Adresse des Quell- und Zielstrings benötigt. Dennoch ist diese Vorgehensweise leicht verständlich. Die Funktion kann über Zeiger jedes Zeichen des Quellstrings einzeln ansprechen (zeichen[x] = *(quelle + x);) und an die Zieladresse weiterleiten (*(ziel + x) = zeichen[x];). Unsere Anweisung gestaltet sich wie folgt:

```
strcpy( &text[2], " !" );
```

Die Anweisung "&text[2]" dürfte nach der vorhergehenden Erklärung keine Schwierigkeiten bereiten, repräsentiert sie doch lediglich die Adresse des dritten Zeichens unseres Strings. Da die zweite Anweisung einen kompletten String darstellt ("" !""), wird nicht ein einzelnes Zeichen des Strings als Adresse,

Abb. 2: Demo-programm 2

```
#include <stdio.h>
#include <string.h>
main()
{
   char text[10];
   text[0] = 'H';
   *(text + 1 ) = 'i';
   strcpy(&text[2]," !");
   printf( "%s", text );
}
```

Nach dem Start gibt das Programm den String "Hi !" auf dem Bildschirm aus.

was übrigens zu unvorhersehbaren Komplikationen bis hin zum Systemabsturz führen würde, sondern die Adresse des ersten Zeichens dieses Strings übergeben. Das Ende des Strings markiert ein NULL-Byte, das von strcpy automatisch angehängt wird.

Neben der statischen Verwaltung von Werten und Strings in Variablen, ist besonders die dynamische Speicherverwaltung, welche sich von "C" aus recht einfach verwenden läßt, interessant. Eben diese gestattet es, nicht von vorne herein auf eine bestimmte Menge Variablenspeicher angewiesen zu sein. Eine dynamische Organisation des Speichers erlaubt, immer genau soviel RAM zu reservieren, wieviel im Augenblick benötigt wird. Dabei gestaltet sich ein Erweitern, bzw. Vermindern des reservierten Speichers generell unproblematisch. An dieser Stelle ein konkretes Beispiel: Unser Problem soll es sein, eine unbekannte Menge Text zeilenweise einzulesen und zwischenzuspeichern. Dabei sei nur bekannt, daß maximal 100 Zeilen übergeben werden, die Länge der einzelnen Zeilen mag beliebig variieren.

Abb. 3: Primitivlösung zum Einlesen bis max. 100 Textzeilen

```
#include <stdio.h>
main()
{
   char Text[100][80];
   (weiterer Programmcode, welcher die
   erste Textzeile einlesen wird)

     strcpy( Text[1], (eingelesener Text) );
   (weiterer Programmcode)
}
```

Die Primitivstlösung unseres Beispiels könnte ein Anfänger wie folgt mit der Überlegung realisieren, daß eine Textzeile "in der Regel" nie mehr als 80 Buchstaben umfaßt:

Die Nachteile liegen auf der Hand: Ersteinmal wird niemand dafür garantieren, daß die Länge einer Textzeile keine 80 Zeichen übersteigen wird; darüberhinaus wird, selbst wenn die Textquelle einmal nur zwei oder noch weniger Zeilen umfaßt, unser Programm gigantische Mengen an Speicher brachliegen lassen. Die logische Folge: Eine speicherfreundlichere, dynamische Lösung hat diese indiskutable Konzeption zu verdrängen. Dabei wäre folgende Vorgehensweise die wohl auf den ersten Blick naheliegendste:

Der Programmierer reserviere ein CHAR-Pointer-Feld mit hundert möglichen Einträgen (char *TextZeiger[100]). Nun erlaubt der MALLOC-Befehl, für die einzelnen Einträge den Speicherplatz zu reservieren, welchen die entsprechende Textzeile benötigt (TextZeiger[x] = (char*) malloc(strlen("(Zeichenkette)"))). Dabei wären bei geringer Zeilenanzahl des Quelltextes nur 100 char-Zeiger umsonst gewesen und der Speicherplatzverlust hätte sich in erträglichen Bahnen bewegt. Dennoch demonstriert diese Vorgehensweise Bereitschaft zum Pfusch und sollte deshalb in keinem Programm, jedenfalls nicht in dieser Form, Anwendung finden. Doch nun, nach all diesen wenig versprechungsvollen Versuchen, zu einem passablen Endergebnis.

Nehmen wir an, unser Programm kam zu der Feststellung, daß zwanzig neue Textzeilen mit einer Länge à 40 Zeichen im Speicher untergebracht werden wollen. Die speicherplatztechnisch und ästhetisch günstige Lösung gestaltet prinzipiell recht einfach. Das Programm hat lediglich den Speicher für zwanzig char-Pointer - jeder dieser Zeiger wird später auf eine Textzeile gerichtet sein - zu reservieren und für jeden dieser char-Pointer einundvierzig Zeichen (einschließlich Nullmarke) zu belegen. Dieser Lösungsweg demonstriert den sinnvollen und unumgänglichen Einsatz von Doppelzeiger (s. Abbildung 4). Bevor uns die Probleme des nächsten Themas belasten sollen, werden wir noch kurz die Verwendung dynamischer Strukturen an-

schneiden. Ein typisches Beispiel dafür ist die Adressenverwaltung. Für eben diese wird der Programmierer eine Struktur entwerfen, welche Name, Adresse, Wohnort und andere Daten über interessante Persönlichkeiten faßt. Wie in den oberen Zeilen ausgeführt, belegt unser Programm nur soviel Speicherplatz wie unbedingt notwendig. Anfangs soll ein einziger Eintrag ausreichen, ein MALLOC-Befehl genügt also, um den notwendigen Speicherplatz anzufordern. Bei Neueintragungen hat der REALLOC-Befehl die Aufgabe, neuen Speicher für unsere Daten zu reservieren. Folgen Sie beim Programmieren der eben vorgestellten Beschreibung, begehen Sie eine grobe Unachtsamkeit. Sollte beispielsweise nach der 678. Eintragung (selbstverständlich wurde in der Zwischenzeit nichts gesichert) der Speicherplatz nicht mehr ausreichen, was bei einer groben Zerstückelung des Hauptspeichers durchaus im Bereich des Möglichen liegen kann, so enthielte eine Anweisung wie

```
Zeiger = realloc( ... );
```

einen Nullzeiger - und die mühsam abgetippten Daten wären in den unendlichen Weiten des Computer-RAMS verschollen. Hier gilt zu beachten, daß bei ungenügenden Speichervorräten ein REALLOC-Befehl die vorhandenen Daten nicht löscht. Es genügt demnach, über einen Hilfszeiger den Speicherplatz zu erweitern und diesen nur im Erfolgsfalle (Hilfszeiger != NULL) zu übernehmen.

Ein aus Platzgründen abgemagertes Beispiel für diese Problematik finden Sie in Abbild 5. Dieses Programm belegt in den ersten Zeilen Speicher für eine Struktur, welche aus einem 20-Byte-String und einem CHAR-Pointer besteht. Über den Pointer werden zusätzlich 100 Byte belegt, in welchen der interessierte Anwender einen Kommentar oder ähnlichen Text ablegen mag. In den nächsten Zeilen finden Sie die praktische Umsetzung der oben aufgeführten Überlegungen; bevor über REALLOC weiterer Speicherplatz für unsere Daten belegt wird, übernimmt eine Hilfsvariable (HilfsZeiger) den Adressenwert des Originalzeigers (ZeigerAdresse). Diese Vorgehensweise garantiert, daß auch bei zu geringen Speicherreserven die alten Daten erhalten bleiben. Sollte der Speicher tatsächlich nicht ausreichen, enthält die zuvor eingeführte Hilfsvarible nach dem Aufruf des REALLOC-Kommandos einen NULL-Zeiger,

```
#include <stdio.h>
#include <string.h>
main()
{
   char **Text;
   int  HilfsVar;
   Text = (char**)
   calloc( 20, sizeof( char*) );
   for( HilfsVar = 0;
        HilfsVar < 20;HilfsVar++)
   {
     *( Text + HilfsVar ) =
     (char*) malloc( 41 );
     strcpy( *( Text+HilfsVar ),
     "(Text)" );
   }
}
```

Abb. 4: Endlich ein passabler Lösungsweg zum dynamischen Verwalten von Text

```
#include <stdio.h>
#include <string.h>
main()
{
   typedef struct AdrFeld
   {
     char Name[20];
     char *Kommentar;
   } AdressFeld;
   AdressFeld *ZeigerAdresse;
   AdressFeld *HilfsZeiger;
   ZeigerAdresse = (AdressFeld*)
   malloc( sizeof ( AdressFeld ) );
   strcpy( ZeigerAdresse->Name,
   "Andreas Kunz" );
   ZeigerAdresse->Kommentar
                  =(char*) malloc( 100 );
   HilfsZeiger = ZeigerAdresse;
   HilfsZeiger = (AdressFeld*) realloc
   (HilfsZeiger, 2 * sizeof(AdressFeld));
   /* Ende, falls nicht genug Speicher */
   if ( HilfsZeiger == NULL ) exit(0);
   ZeigerAdresse = HilfsZeiger;
   strcpy( (ZeigerAdresse+1)->Name,
   "Karl Kurz" );
   (ZeigerAdresse+1)->Kommentar =
   (char*) malloc( 100 );
}
```

Abb. 5: Prinzipielle Verwendung dynamisch angelegter Strukturen

Abb. 6: Beispiel für Zeiger auf Funktionen

```
#include <stdio.h>
void Dummy( void )
{
}
void Hallo( void )
{
   printf("\n\rHallo");
}
void Ade( void )
{
   printf("\n\rAde");
}
void Aufruf( void ( *Dummy )( void ) )
{
   Dummy();
}
main()
{
   Aufruf( Hallo );

   Aufruf( Ade );
}
```

Das Programm gibt die beiden Strings "Hallo" und "Ade" aus.

der Originalzeiger bleibt jedoch unverändert und wird logischerweise nicht durch den Wert des Hilfszeigers ersetzt, wären dadurch doch alle Daten verloren. Unser Programm reagiert auf einen ungenügenden Speichervorrat mit einem Abbruch des Programmes. In der Praxis wäre eine Notiz am Bildschirm oder ähnliches natürlich angebrachter, soll hier aber nicht weiter ausgeführt werden; das Beispiel hat nur die Aufgabe die grundsätzliche Vorgehensweise zu erläutern.

Sollen Sie in der Zeiger-Thematik noch nicht ganz firm sein, wird Ihnen die Belegung der dynamisch angelegten Variablen ein wenig seltsam vorkommen. Es ist kein Sternchen weit und breit zu sehen, das auf die Verwendung von Zeigern hindeuten könnte. Dieser Einwand scheint verständlich, wäre es doch viel logischer, nehmen wir an, das Programm hätte Speicher für zwei unserer Adreßfelder belegt, die Felder wie folgt zu belegen:

```
strcpy( (*ZeigerAdresse).Name, "Andreas Kunz" );
strcpy( (*(ZeigerAdresse + 1)).Name,
"Karl Kurz" );
```

Diese Vorgehensweise entspräche der konventionellen Zeigerverwendung und ist durchaus gestattet. Doch schon ein kurzer Blick offenbart die Undurchsichtigkeit dieser Zeilen. Aus diesem Grund kennt "C" eine weitere Methode, welche das Problem eleganter und sauberer lösen läßt:

```
strcpy( ZeigerAdresse->Name, "Andreas Kunz" );
strcpy( (ZeigerAdresse + 1)->Name, "Karl Kurz" );
```

Der Strukturverweis-Operator ("->") genügt "C" schon, um zu wissen, daß es sich um eine dynamisch angelegte Struktur handelt. Die Kette von Beispielen ließe sich an dieser Stelle natürlich noch beliebig ins Abstrakte fortsetzen. Dennoch dürfte die prinzipielle Vorgehensweise geklärt und Sie auch für scheinbar unüberwindliche Probleme gerüstet sein.

Das letzte Thema dieses Artikels wird sich mit dem Problem beschäftigen, eine praktische Anwendung für Zeiger auf Funktionen zu suchen. Dieses Thema kann beispielsweise beim Programmieren einer Druckertoolbox akut werden. Stellen Sie sich vor, die Toolbox sollte dem User die Möglichkeit bieten, an einer bestimmten Zeile eine von im angegebene Funktion aufrufen zu können. In diesem Fall hat es verständlicherweise keinen Sinn, den Käufer lediglich darauf hinzuweisen, die gewünschten Zeilen in den Toolbox-Sourcecode einzubauen. Jeder fortgeschrittene Programmierer würde ein derartiges Produkt schmunzelnd beiseite legen.

Abbildung 6 zeigt die prinzipielle Vorgehensweise. Wichtig ist dabei, daß eine Dummyfunktion vorhanden sein muß, welche den Typ einer später übergebenen Funktion charakterisiert. Aus dieser Vorgehensweise ist ersichtlich, daß immer nur ein bestimmter Funktionstyp übergeben kann, das bedeutet, die Art der Parameter muß in jedem Falle übereinstimmen (in unserem Beispiel werden keine Parameter übergeben). Da weder Handbücher, noch Sekundärliteratur näher die Verwendung und Arbeitsweise von Zeigern ausführt, hoffen wir, Ihnen mit diesen Seiten wichtige Tips und Anregungen für zukünftige Projekte gegeben zu haben.