Dynamische Speicherplatzreservierung

Eindimensionale Arrays

Bei Anwendungen, die Arrays verwenden, steht man häufig vor dem Problem, daß die Größe (Anzahl der Elemente) von Arrays zum Zeitpunkt der Übersetzung des Programms noch nicht feststeht, sondern sich erst während der Ausführung durch Benutzereingabe oder auf Grund der Daten ergibt. Sowohl in F als auch in C gibt es Mechanismen, um zur Laufzeit von Programmen dynamisch Speicherplatz anzufordern (zu "allokieren") und wieder freizugeben. Am einfachsten ist dies bei eindimensionalen Objekten (Vektoren), wo auch die Ähnlichkeit der beiden Sprachen am größten ist.

Die Verwendung dynamisch reservierter Arrays zerfällt in drei Teilaufgaben:

- Deklaration eines Objekts als dynamisch reservierbar (diese Eigenschaft muß zur Compilationszeit bekannt sein)
- Eigentliche Reservierung des Speicherplatzes (danach kann das Objekt wie ein gewöhnlicher Array verwendet werden)
- Freigabe des Speicherplatzes (entweder explizit oder automatisch am Programmende).

In F werden dynamisch allokierbare eindimensionale (und analog mehrdimensionale) Arrays durch ein "leeres" `dimension-` und das `allocatable`-Attribut bei der Variablendeklaration gekennzeichnet

type`,dimension(:),allocatable::`*variable_list*

Wollte man also z.B. drei eindimensionale Arrays `x`, `y` und `z` vom Typ `real` und mit noch unbekannter Anzahl von Elementen definieren, so könnte das mit der Deklaration

```
real,dimension(:),allocatable::x,y,z
```

erfolgen. Diese Variablen dürfen aber noch nicht verwendet werden, solange für sie nicht explizit Speicherplatz angefordert wurde.

Die eigentliche Reservierung von Speicherplatz erfolgt mit dem `allocate`-Befehl

`allocate(`*array_1*`(`*dim_1*`)[,`*array_2*`(`*dim_2*`),...][,stat=`*status*`])`

Dabei beschreiben *dim_1*, *dim_2*, ... die Indexbereiche der Arrays, und *status* ist eine Statusvariable vom Typ `integer`, deren Wert anzeigt, ob die Reservierung erfolgreich war (*status*$= 0$) oder nicht (*status*$\neq 0$). Die Speicherplatzreservierung für die Vektoren `x`, `y` und `z` von vorhin könnte also

```
allocate(x(100),y(0:20),z(-15:15))
```

lauten, wenn man auf den Statustest verzichtet.

Werden Arrays nicht mehr benötigt, so kann ihr Speicherplatz mit `deallocate` explizit freigegeben werden

`deallocate(`*array_1*[`,`*array_1*`,...`]`)`

also z.B.

```
deallocate(y,z)
```

Es spricht übrigens nichts dagegen, ein deallokiertes Objekt neuerlich (u.U. mit einer anderen Anzahl von Elementen) zu allokieren.

In C werden dynamisch reservierbare eindimensionale Arrays am übersichtlichsten behandelt, indem man sie zunächst als Pointer deklariert und dann "vergißt", daß es eigentlich Pointer sind. Ein Pointer ist eine Variable, deren Inhalt nicht direkt einen Wert, sondern dessen Speicheradresse darstellt, d.h. sie "zeigt" auf die Adresse, unter der dieser Wert im Speicher zu finden ist—in unserem Fall wird das die Adresse des ersten Array-Elements sein. In Deklarationen werden Pointer dadurch gekennzeichnet, daß man einen Stern (`"*"`) vor ihren Namen setzt und ihnen den Typ der Werte gibt, auf die sie zeigen

type `*`*array_1*[`,*`*array_1*`,...];`

Sollen analog zum Fall von F drei `float`-Arrays `x`, `y` und `z` deklariert werden, dann kann das z.B. so aussehen

```
float *x,*y,*z;
```

Zur eigentlichen Speicherplatzreservierung dienen die Funktionen `malloc`

array`=(`*type*`*)malloc(`*num_bytes*`);`

oder `calloc`

array`=(`*type*`*)calloc(`*num_elements*`,`*size_element*`);`

wo *num_bytes* die absolute Größe des Arrays in Bytes ist, bzw. *num_elements* die Anzahl der zu reservierenden Array-Elemente und *size_element* die Größe eines Array-Elements (in Bytes) angeben. Dadurch wird ein zusammenhängender Bereich im Speicher angefordert, der groß genug ist, alle Elemente von *array* aufzunehmen. Obwohl es sich im Prinzip um ganze Zahlen handelt, sind alle Argumente formal vom Typ `size_t` und müssen/können bei Bedarf explizit durch einen `cast`-Operator konvertiert werden. Bei Verwendung von `calloc` statt `malloc` wird zusätzlich noch der reservierte Speicherplatz mit Nullen initialisiert. Wenn die Speicherplatzreservierung erfolgreich war, ist der Rückgabewert von `malloc`/`calloc` ein Pointer vom Typ `void` auf die erste Speicheradresse des reservierten Bereichs. Diese Adresse wird der Pointervariablen *array* (die auf das erste Array-Elemnent zeigen soll) zugewiesen, nachdem sie mit Hilfe eines `cast` auf den Typ von *array* konvertiert wurde. Ist die Reservierung nicht erfolgreich, so wird statt einer gültigen Speicheradresse der Wert `NULL` zurückgegeben. Den Speicherplatzbedarf eines einzelnen Array-Elements (in Bytes) kann man mit der Funktion

`sizeof(`*type*`)`

abfragen. Die Übersetzung des obigen Fortran-Beispiels nach C könnte damit folgendermaßen aussehen

```
x=(float*)malloc(100*sizeof(float));
y=(float*)malloc(21*sizeof(float));
z=(float*)malloc(31*sizeof(float));
```

(ebenfalls ohne Erfolgstest).

Die Rückgabe des für *array* reservierten Speicherplatzes erfolgt in C mit dem Befehl

```
free(array);
```

also z.B.

```
free(y);
free(z);
```

Um in Ausdrücken oder Zuweisungen auf die Werte individueller Array-Elemente zuzugreifen, könnte man sich des bei der Deklaration von Pointern erwähnten `"*"`-Operators bedienen: Ist nämlich p ein Pointer, so bekommt man mit `*`p den Wert, der an der Speicherstelle steht, deren Adresse die (Pointer)-Variable p enthält. Analog liefert `*(`p`+1)` den an der nächsten, `*(`p`+2)` den an der übernächsten Adresse gespeicherten Wert, usw. Man könnte daher die einzelnen Elemente von *array* mit `*`*array*, `*(`*array*`+1)`, `*(`*array*`+2)` usw. ansprechen. Glücklicherweise gibt es dafür die äquivalente Notation *array*`[0]`, *array*`[1]`, *array*`[2]`, ..., d.h. *array*`[i]` ist der `i`-te Wert hinter der Startadresse von *array*. Damit ist, abgesehen von der Deklaration und Allokation/Deallokation, die Verwendung dynamisch erzeugter Arrays weitgehend identisch mit der von statischen Arrays (mit zur Compilationszeit fixierter Größe). So können im obigen Beispiel `x`, `y` und `z` als Vektoren mit den Elementen `x[0]` bis `x[99]`, `y[0]` bis `y[20]` und `z[0]` bis `z[30]` verwendet werden—ganz so, als ob man sie statisch mittels

```
float x[100],y[21],z[31];
```

deklariert hätte.