Unterprogramme, Pointer und die Übergabe von Arrays

Unterprogramme

Wie schon im Abschnitt über Funktionen erwähnt, versteht man unter einem Unterprogramm im eigentlichen Sinn eine Prozedur, welche die Werte der an sie übergebenen Variablen verändern darf. Im Gegensatz zu Funktionen, die nur einen einzigen Wert (eben den "Funktionswert") an das rufende Programm zurückliefern können (F kennt allerdings auch Array-wertige Funktionen), wird man also ein Unterprogramm dann verwenden, wenn das Ergebnis einer Berechnung aus mehreren Werten besteht. Es muß in der Parameterliste des Unterprogramms nur eine entsprechende Anzahl von Variablen zur Aufnahme dieser Werte vorgesehen werden. Zusätzlich kann ein Unterprogramm auch noch die Werte globaler Variablen verändern.

Während C nicht zwischen Funktionen und Unterprogrammen unterscheidet (alle Prozeduren sind Funktionen), heißt in F ein Unterprogramm *Subroutine* und hat die folgende Gestalt

```
subroutine subroutine_name(argument_list)
  type,intent(in)::arg_sublist_1
  ...
  type,intent(out)::arg_sublist_2
  ...
  type,intent(inout)::arg_sublist_3
  ...
  local declarations
  ...
  executable statements
  ...
  return
end subroutine subroutine_name
```

Das Attribut `intent` bei der Deklaration der Argumente des Unterprogramms kennzeichnet, welche Parameter von der Prozedur nur gelesen (`in`), nur geschrieben (`out`) oder sowohl gelesen als auch geschrieben (`inout`) werden dürfen. Falls die Parameterliste *argument_list* leer ist, kann mit dem rufenden Programm nur über globale Variablen kommuniziert werden. Für diese muß aber im Unterprogramm kein `intent` definiert werden.

Unterprogramme müssen, ebenso wie Funktionen, in F in einen `module`-Block eingeschlossen werden, und der Name des Unterprogramms muß global sichtbar gemacht werden:

```
module module_name
  public::subroutine_name
  ...
contains
  subroutine subroutine_name(...)
    ...
  end subroutine subroutine_name
  ...
end module module_name
```

Der Aufruf eines Unterprogramms erfolgt mit dem Befehl `call`

`call` *subroutine_name*`(`*argument_list*`)`

Da in F Parameter per Adresse (call by name) übergeben werden, genügt es, in der Argumentliste einfach die Namen der zu übergebenden Variablen anzuführen. Das Unterprogramm greift direkt auf die Speicherplätze dieser Variablen zu, und alle an ihnen vorgenommenen Veränderungen sind im rufenden Programm unmittelbar sichtbar.

Pointer

In C sind alle Prozeduren Funktionen. Also muß man, um den Mechanismus von Unterprogrammen zu implementieren, Funktionen schreiben, die den Wert eines oder mehrerer Argumente verändern können. Da in C aber Argumente per Wert (call by value) übergeben werden, kann dies nur dadurch geschehen, daß man für jene Variablen, die vom Unterprogramm verändert werden sollen, nicht deren Wert, sondern *ihre Adresse im Speicher* übergibt. Dann kann, wie im Fall von F, das Unterprogramm direkt mit diesen Speicherinhalten rechnen, und alle Änderungen wirken sich automatisch im rufenden Programm aus.

(Da solche Unterprogramme keinen Funktionswert im eigentlichen Sinn liefern, kann man ihnen den Typ `void` geben. Häufig definiert man sie auch als Typ `int` und gibt als "Funktionswert" einen ganzzahligen Code zurück, der anzeigt, ob im Unterprogramm Fehler aufgetreten sind oder nicht.)

Die Adresse einer Variablen im Speicher liefert der *Adreßoperator* `"&"`, der dem Namen der Variablen vorangestellt wird

`&x` = Adresse der Variablen `x`

Diese Konstruktion wird z.B. beim Aufruf der `scanf-`"Funktion" verwendet, wenn von der Tastatur die Werte von `x` und `y` eingelesen werden sollen

```
scanf("%f,%f",&x,&y);
```

Hier werden also die Adressen von `x` und `y` an das Unterprogramm übergeben, das damit unmittelbar auf die dort liegenden Speicherinhalte Zugriff hat und von sich aus mit den eingelesenen Werten überschreiben kann.

Um im Unterprogramm mit diesen Adressen arbeiten zu können, braucht man Variablen zur Speicherung von Adressen. Solche Variablen nennt man *Pointer*, weil sie auf den Speicherplatz bzw. den dort gespeicherten Wert "zeigen". (Pointer gibt es sowohl in F als auch in C, doch braucht man in F Pointer eher selten, und auch in C führt die sparsame Verwendung von Pointern zu lesbarerem und sichererem Code.)

Obwohl Speicheradressen natürlich ganzzahlig sind, haben Pointer in C einen eigenen Typ, der sich nach dem Typ des Wertes richtet, auf den der Pointer zeigt, d.h. es gibt "Pointer auf `int`", "Pointer auf `float`" usw. Pointer müssen deklariert werden wie andere Variablen auch

type `*`*pointer_1*`,*`*pointer_2*`,...;`

Dabei ist *pointer_1*, *pointer_2*, ... der Name des Pointers und *type* der Typ des Wertes, auf den der Pointer zeigt.

Den Wert an der Speicherstelle, auf die ein Pointer zeigt, erhält man mit dem *Indirektionsoperator* `"*"`, der dem Namen des Pointers vorangestellt wird

`*p` = an der Adresse `p` gespeicherter/zu speichernder Wert

Mit den so angesprochenen Speicherinhalten kann man rechnen wie mit gewöhnlichen Variablen, d.h. sie können auf beiden Seiten von Zuweisungen vorkommen, z.B.

```
float *xp,*yp,*zp;
float u,v,w;
...
xp=&u;
yp=&v;
zp=&w;
...
u=*yp+w;
*zp=*xp+v;
...
```

Um also in C ein Unterprogramm zu schreiben, trennt man die Argumente in solche, die von der Prozedur nur gelesen [entsprechend `intent(in)` von F], und solche, die auch gesetzt werden [entsprechend `intent(out)` oder `intent(inout)`], und übergibt die ersteren per Wert, die letzteren aber per Adresse (Pointer), z.B.

```
void sub(float rr1,float rr2,...,float *ww1,float *ww2,...) {
  ...
  *ww1=...;
  *ww2=...;
  ...
}
...
int main(...) {
  ...
  float r1,r2,...,w1,w2,...;
  ...
  sub(r1,r2,...,&w1,&w2,...);
  ...
}
```

Übergabe von (mehrdimensionalen) Arrays in F

Assumed-Shape Arrays

Das allgemeine Problem bei der Übergabe von mehrdimensionalen Arrays an Unterprogram-

me besteht darin, daß alle (also auch mehrdimensionale) Arrays *linear* auf den Speicher abgebildet werden. So werden in F etwa die Elemente einer Matrix `a(m,n)` spaltenweise fortlaufend gespeichert

```
a(1,1), ..., a(m,1), a(1,2), ..., a(m,2), ...,a(1,n), ..., a(m,n)
```

d.h. das Element `a(i,j)` steht `i+(j-1)*m-1` Positionen hinter der Startadresse (und wird auch mittels "Startadresse plus Offset" angesprochen). Um eine an ein Unterprogramm übergebene Matrix korrekt adressieren zu können, muß das Unterprogramm also neben der Startadresse von `a` auch noch die Spaltenlänge `m` kennen.

Die Übergabe von Arrays an Unterprogramme ist in F aber dennoch sehr einfach: Es muß nur die Startadresse (d.h. der Name) des Arrays explizit als Parameter übergeben werden, die Informationen über die Dimensionen des Arrays werden dann automatisch übertragen (aber nicht die Startwerte der Indizes, wenn diese nicht bei 1 beginnen). In den Deklarationen der Unterprogramm-Parameter kann man daher die Dimensionen solcher Arrays nach dem Muster

```
dimension(:,:)
```

unbestimmt lassen. Diesen Mechanismus nennt man *Assumed-Shape Arrays*.

Die aktuelle Anzahl der Elemente in der Dimension *dimension* eines Arrays *array* kann man im Unterprogramm mit der `size`-Funktion

`size(`*array*`,`*dimension*`)`

abfragen.

Das folgende Beispiel zeigt die Übergabe einer Matrix und eines Vektors, dessen Startindex nicht `1`, sondern `imin` ist

```
subroutine sub1(a,v,imin)
  real,dimension(:,:),intent(in)::a
  real,dimension(imin:),intent(inout)::v
  integer::m,n
  ...
  m=size(a,1)
  n=size(a,2)
  ...
end subroutine sub1
```

Automatic Arrays

Häufig braucht man in Unterprogrammen temporäre Arrays, deren Größe von Aufruf zu Aufruf variieren kann. Solche Objekte muß man in F nicht explizit dynamisch erzeugen, sondern es genügt, sie als *Automatic Arrays* zu deklarieren. Das sind lokale Arrays, deren Dimensionen von den Parametern des Unterprogramms abhängig gemacht werden können.

Im folgenden Beispiel sind `b` und `c` Automatic Arrays

```
subroutine sub2(a,m,...)
  integer,intent(in)::m
  real,dimension(:,:),intent(inout)::a
  real,dimension(2*m)::b
  real,dimension(size(a,1))::c
  ...
end subroutine sub2
```

Übergabe von (mehrdimensionalen) Arrays in C

Bei der Verwendung von Arrays in C treten zwei Schwierigkeiten auf:

- Es besteht eine prinzipielle Inkonsistenz zwischen mehrdimensionalen Arrays fixer Größe und dynamisch erzeugten Arrays. Während die ersteren wie in F linear fortlaufend gespeichert sind (allerdings z.B. Matrizen zeilenweise) und über eine Startadresse plus Offset angesprochen werden, sind die letzteren stets Pointer auf einen Vektor von Pointern auf Objekte niedrigerer Dimensionalität (also im Fall einer Matrix ein Pointer auf einen Vektor von Pointern auf Zeilenvektoren) und daher i.a. *nicht* fortlaufend gespeichert. Ein Unterprogramm kann aber nicht unterscheiden, ob ein Array fix oder dynamisch erzeugt ist.

- Selbst wenn man sich auf Arrays von zur Compilationszeit bekannter (fixer) Größe beschränkt, fehlt ein Mechanismus, um im Unterprogramm z.B. die Zeilenlänge `n` einer als Parameter übergebenen Matrix zu vereinbaren, aus der der Compiler beim Auftreten von Matrixelementen der Form `a[i][j]` den Offset `i*n+j` für das Unterprogramm selbständig berechnen könnte.

Es empfiehlt sich daher, in C ***alle*** *Arrays dynamisch zu erzeugen und die syntaktische Verwandtschaft von Arrays und Pointern auszunützen.*

Diese besteht einerseits darin, daß man, wenn `a` ein Pointer ist, den Wert, der `i` Speicherstellen nach dem Wert steht, auf den der Pointer zeigt, sowohl in der Form `*(a+i)` als auch in der Form `a[i]` ansprechen kann. (Da aus dem Typ des Pointers hervorgeht, wieviel Platz zur Speicherung des Wertes notwendig ist, muß man den Pointer jeweils nur um `1` erhöhen, um auf den nächsten Wert im Speicher zu zeigen, unabhängig von der tatsächlichen Schrittweite in Bytes.) Andererseits ist diese Nomenklatur rekursiv, d.h. wenn `a[i]` kein Wert, sondern wiederum ein Pointer ist, dann kann man den Wert `j` Speicherstellen nach dem Wert, auf den der Pointer bei `a[i]` zeigt, mit `a[i][j]` ansprechen, usw.

Um z.B. eine `m*n` Matrix `a` vom Typ `float` zu erzeugen, kann man folgendermaßen vorgehen: Es wird ein Pointer auf einen Vektor von Pointern vom Typ `float` definiert

```
float **a;
```

Der Speicherplatz für den Vektor von Pointern {`a[0]`, `a[1]`, ..., `a[m-1]`} muß mit `malloc` (oder `calloc`) angefordert werden

```
a=(float**)malloc(m*sizeof(float*));
```

(`m` ist die Anzahl der Zeilen der Matrix). Nun wird für jedes Element `a[i]` des Vektors von Pointern noch der Speicherplatz für die Zeile der Matrix reserviert, auf deren Beginn `a[i]` zeigen soll

```
for(i=0;i<=m-1;i=i+1) {
  a[i]=(float*)malloc(n*sizeof(float));
}
```

(`n` ist die Anzahl der Spalten der Matrix). Die "Matrixelemente" können dann in der üblichen Form `a[i][j]` angesprochen werden.

Das folgende Beispiel zeigt, wie man mit Hilfe dieses Mechanismus relativ einfach eine `m*n` Matrix `a` erzeugen und an ein Unterprogramm übergeben kann. Dabei müssen zwar die Dimensionen der Matrix explizit mitübergeben werden, damit man im Unterprogramm weiß, in welchen Bereichen sich (z.B. in Schleifen) die Indizes `i` und `j` bewegen dürfen; zur Berechnung der Adresse eines Elements `a[i][j]` braucht der Compiler aber *nicht* die Spaltenanzahl `n`, da die Adressen aller Zeilenanfänge in dem Pointer-Vektor `a[i]` stehen, auf den das Unterprogramm direkt zugreifen kann.

```
void sub(float **a,int m,int n,...) {
  int i,j,m,n;
  ...
  a[i][j]=...;
  ...
}
...
int main(...) {
  int i,m,n;
  float **a;
  ...
  a=(float**)malloc(m*sizeof(float*));
  for(i=0;i<=m-1;i=i+1) {
    a[i]=(float*)malloc(n*sizeof(float));
  }
  ...
  sub(a,m,n,...);
  ...
}
```