Wertebereich und Genauigkeit der Zahlendarstellung

Sowohl F als auch C kennen bei ganzen und Floating Point-Zahlen Datentypen verschiedener "Genauigkeit". Bei ganzen Zahlen, die stets exakt dargestellt werden und mit denen auch exakt gerechnet wird, betrifft das nur den möglichen Wertebereich. Je nachdem, wie viele Bytes an Speicherplatz für eine Variable reserviert sind, können darin nur ganze Zahlen bis zu einer bestimmten Maximalgröße untergebracht werden. Beim Standardtyp sind das derzeit meist 4 Bytes (32 Bits), was für ganze Zahlen einen Wertebereich von $-2^{31} = -2147483648$ bis $2^{31} - 1 = 2147483647$ oder ca. $\pm 2 \times 10^9$ ergibt (die genauen Grenzen hängen von den Details der Zahlendarstellung ab). Bei Floating Point-Zahlen wird der Speicherplatz pro Variable noch in Bereiche für die Darstellung des Exponenten und der Mantisse unterteilt. Wie groß diese Bereiche sind, ist implementationsabhängig, auf jeden Fall können darin aber wieder nur Exponenten beschränkter Größe bzw. Mantissen beschränkter Länge untergebracht werden. Bei Speicherung nach dem weitverbreiteten IEEE Standard ergeben sich für 4-Byte Floating Point-Variablen ein Zahlenbereich von etwa 10^{-37} bis 10^{+37} und eine relative Genauigkeit von 6 Dezimalstellen.

Da es einerseits wünschenswert sein mag, bei Programmen mit großem Speicherplatzbedarf den Speicherplatz pro Variable zu reduzieren, andererseits für numerisch heikle Anwendungen Wertebereich oder relative Genauigkeit der Standard-Datentypen nicht ausreichend sein können, kennt C u.a. die folgenden Datentypen für ganze

```
char
short
int
long
```

bzw. Floating Point-Zahlen

```
float
double
```

Dabei ist `char` zwar kein ganzzahliger Typ im engeren Sinn, definiert aber eine konkrete Speicherplatzgröße, nämlich eine 1-Byte Variable oder Konstante. Bei allen anderen Datentypen sind Speicherplatzbedarf und damit Wertebereich und/oder Genauigkeit implementationsabhängig.

Um herauszufinden, wieviel Speicherplatz für eine Variable eines bestimmten Typs reserviert wird, kann man den `sizeof`-Operator verwenden.

`sizeof(`*expression*`)`

liefert die Anzahl der Bytes, die zur Darstellung von *expression* verwendet werden, z.B.

```
short i;
...
printf("i occupies %d bytes\n",sizeof(i));
```

In gemischten numerischen Ausdrücken werden bei Bedarf Variablen oder Konstante einfacheren automatisch in solche komplizierteren Typs (also z.B. `int` in `float`) umgewandelt. Explizit kann man eine Konversion durch den Cast-Operator

(*type*) *expression*

erzwingen. So haben im folgenden Beispiel beide Anweisungen denselben Effekt

```
float x;
...
x=1/3.0;
x=1/((float)3);
...
```

Die Definition numerischer Nicht-Standardtypen in F ist wesentlich portabler (wenn auch auf den ersten Blick etwas undurchsichtig). Dabei ist man von der Überlegung ausgegangen, daß es für numerisch anspruchsvolle Anwendungen nicht essentiell ist anzugeben, wie viele Bytes eine Variable belegen soll, sondern es ist viel wichtiger, für die Variablen *maschinenunabhängig* eine Minimalgenauigkeit bzw. einen minimalen Wertebereich verlangen zu können. Nur dadurch kann garantiert werden, daß eine umfangreichere Berechnung überhaupt zu einem sinnvollen Endergebnis führt.

Die Definition von Nicht-Standardtypen erfolgt mit Hilfe des zusätzlichen Parameters `kind` bei der Variablendeklaration

numeric_type`(kind=`*kind*`)::`*variable_list*

`kind` kann nichtnegative ganzzahlige Werte annehmen, aber welche Werte erlaubt sind und was sie im einzelnen bedeuten, ist von Maschine zu Maschine verschieden. Um portable Programme zu schreiben, muß man daher zuerst herausfinden, welcher `kind`-Wert die Minimalforderungen an Genauigkeit und/oder Wertebereich erfüllt. Dazu dienen die Funktionen

`selected_int_kind(`*range*`)`
`selected_real_kind(`[*precision*][,*range*]`)`

Dabei ist *range* der maximale dezimale Exponent der darzustellenden ganzen oder Floating Point-Zahlen und *precision* (nur bei Floating Point-Zahlen) die Anzahl der signifikanten dezimalen Stellen. (*range*=2 heißt also für ganze Zahlen, daß Werte von -100 bis $+99$ möglich sein sollen.) Diese Funktionen liefern den `kind`-Wert des einfachsten Datentyps zurück, der den Anforderungen genügt, bzw., wenn ein solcher nicht existiert, einen negativen Wert.

Die `kind`-Werte dürfen allerdings nicht als Literals angegeben werden, sondern müssen benannte Konstante sein. Das folgende Beispiel zeigt die Deklaration von "kurzen" ganzzahligen Variablen mit Maximalwerten im Bereich ± 1000 sowie reellen und komplexen Floating Point-Zahlen mit mindestens 12 Dezimalstellen Genauigkeit:

```
integer,parameter::short=selected_int_kind(3)
integer,parameter::double=selected_real_kind(12)
...
integer(kind=short)::i
real(kind=double)::x
complex(kind=double)::z
...
```

Um umgekehrt herauszufinden, welche Eigenschaften der Datentyp eines Ausdrucks *expression* auf einer konkreten Maschine wirklich hat, stehen u.a. die Funktionen

`kind(`*expression*`)`	`kind`-Wert
`bit_size(`*expression*`)`	Anzahl der Bits in der Darstellung (ganzzahlig)
`range(`*expression*`)`	dezimaler Exponentenbereich
`precision(`*expression*`)`	Anzahl signifikanter Dezimalstellen (Floating Point)
`huge(`*expression*`)`	größtmöglicher Wert

zur Verfügung.

Literals eines bestimmten Typs kennzeichnet man durch Anhängen des für den `kind`-Wert gewählten Namens, z.B. mit den Definitionen des obigen Beispiels

```
i=365_short
x=0.123456789_double
```

Bei expliziter Umwandlung eines Datentyps in einen anderen kann bei Konversionsfunktionen wie `int` oder `real` der `kind`-Wert als optionaler weiterer Parameter in der Form

```
int(expression[,[kind=]kind])
real(expression[,[kind=]kind])
...
```

angegeben werden. Mit den obigen Definitionen also z.B.

```
i=int(365.0,kind=short)
```

Arrays

Grundlegendes

Die bisher betrachteten Datentypen waren "primitive", also in physikalisch-mathematischer Sprechweise skalare Objekte. Daneben gibt es in fast allen Programmiersprachen aber natürlich auch die Möglichkeit, höherdimensionale Objekte, also Vektoren, Matrizen usw. zu definieren. Der allgemeine Ausdruck dafür ist "Arrays" oder "Felder".

Die Deklaration, Darstellung und Verwendung von Arrays ist in F und C teilweise recht ähnlich, teilweise aber auch sehr verschieden. Zu den wesentlichen Unterschieden gehört einerseits, daß in F Arrays als eigene Objekte auch im ganzen angesprochen und manipuliert werden können (s. später). Im Gegensatz dazu kann man sich in C immer nur auf einzelne Elemente eines Arrays beziehen; es müssen also stets Indizes und Schleifen verwendet werden.

Ein weiterer gravierender Unterschied besteht in der Form, in der mehrdimensionale Objekte gespeichert werden. Da praktisch alle Computer mit einem eindimensionalen Speichermodell arbeiten, müssen höherdimensionale Objekte auf irgendeine Weise auf einen eindimensionalen Adreßraum abgebildet werden. In F erfolgt das so, daß z.B. die Elemente einer Matrix

$$\begin{matrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{matrix}$$

auf folgende Weise

$$a_{1,1}\ a_{2,1}\ \ldots\ a_{m,1}\quad a_{1,2}\ a_{2,2}\ \ldots\ a_{m,2}\quad \ldots\quad a_{1,n}\ a_{2,n}\ \ldots\ a_{m,n}$$

also spaltenweise gespeichert werden. Im Gegensatz dazu werden in C Matrizen zeilenweise gespeichert, d.h. die Matrix

$$\begin{array}{cccc} a_{0,0} & a_{0,1} & \ldots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \ldots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \ldots & a_{m-1,n-1} \end{array}$$

wird im Speicher in der Form

$$a_{0,0}\ a_{0,1}\ \ldots\ a_{0,n-1}\quad a_{1,0}\ a_{1,1}\ \ldots\ a_{1,n-1}\quad \ldots\quad a_{m-1,0}\ a_{m-1,1}\ \ldots\ a_{m-1,n-1}$$

abgebildet. Auf diesen Unterschied muß insbesondere Rücksicht genommen werden, wenn in einer aus F und C gemischten Anwendung Unterprogramme in F Arrays an Unterprogramme in C übergeben oder umgekehrt.

Die Wahl der Indizes in den obigen Matrizen hat gleichzeitig noch einen weiteren Unterschied zwischen F und C illustriert: Während in F Array-Indizes beliebige Intervalle aus der Menge der ganzen Zahlen (also auch negative Werte) durchlaufen können, müssen in C Indizes stets mit 0 beginnen.

Deklaration und Verwendung

Es werden zunächst nur Arrays betrachtet, deren Größe zur Compilationszeit vorgegeben ist. Darüber hinaus können aber sowohl in F als in C Arrays auch zur Laufzeit dynamisch erzeugt werden.

Arrays werden in F durch den `dimension`-Parameter der Variablendeklaration definiert

type`,dimension(`*dim_1*`,`*dim_2*`,...)::`*variable_list*

wobei *dim_1*, *dim_2*, ... die Indexbereiche der einzelnen Dimension des Arrays beschreiben und von der Form

i_min`:`*i_max*

oder

i_max

sind. Der Index kann jeweils die Werte von *i_min* bis *i_max* bzw., wenn *i_min* nicht angegeben ist, von 1 bis *i_max* annehmen. Beispielsweise definiert

```
real,dimension(10)::v
real,dimension(10,20)::mat
real,dimension(-3:6)::u
```

zwei Vektoren `u` und `v` mit je 10 Elementen und eine 10×20 Matrix `mat`. Bei `u` sind Indexwerte von -3 bis 6 vorgesehen, bei `v` von 1 bis 10; bei `mat` läuft der erste Index von 1 bis 10, der zweite von 1 bis 20.

Die Deklaration von Arrays in C erfolgt durch Angabe der Anzahl der Elemente pro Dimension, *dim*, nach dem Variablennamen

type name`[`*dim_1*`][`*dim_2*`]...;`

Der Index kann dann jeweils die Werte 0, 1, ..., $dim-1$ annehmen. Die zu dem vorigen Beispiel analogen Deklarationen könnten also lauten

```
float u[10],v[10];
float mat[10][20];
```

Arrays können—wie skalare Variable auch—bei der Deklaration mit Werten initialisiert werden. In F benützt man dazu einen speziellen Array-Konstruktor, der einen Vektor von Ausdrücken erzeugt, in C eine Liste. Wollte man z.B. einen (konstanten) Vektor `primes` mit den ersten fünf Primzahlen definieren und bei der Deklaration gleich initialisieren, so könnte das in F durch

```
integer,dimension(5),parameter::primes=(/2,3,5,7,11/)
```

geschehen und in C durch

```
const int primes[5]={2,3,5,7,11};
```

Die zusätzlichen Modifikationen `parameter` bzw. `const` in diesem Beispiel besagen, daß es sich im speziellen Fall nicht um Arrays von Variablen, sondern von benannten Konstanten handeln soll.

Bei der Verwendung von Array-Elementen in numerischen Ausdrücken sowie auf der linken Seite von Zuweisungen, d.h. überall, wo auch skalare Variable stehen könnten, werden der oder die entsprechenden Indizes in runden (F) bzw. eckigen [C] Klammern angegeben. Zum Beispiel in F

```
integer i,j
real,dimension(10)::u,v
real,dimension(10,20)::mat
...
mat(1,1)=0.0
...
u(i)=mat(i,2*j+1)*v(j)+1.0
...
```

und ähnlich in C

```
int i,j;
float u[10],v[10];
float mat[10][20];
...
mat[0][0]=0.0;
...
u[i]=mat[i][2*j+1]*v[j]+1.0;
...
```