Chapter 9
Other Data Types

Many computers use 32 bits to store integers and real numbers. So typically, the values of an integer is in the range -2147483647 to 2147483647 and a real number is in the range -10^{38} to 10^{38}.

Sometimes, we need more precision of numbers. Fortran 90 provided parameterized types.

`type(KIND = kind-number)` for example:

```
REAL(KIND = 2) ::  x
```

```
INTEGER(KIND = 8) ::  I
```

The number of `kind-number` varies from one Fortran complier to another. Fortran provides some intrinsic functions for use with parameterized types.

`SELECTED_REAL_KIND(p, r)`

where `p` and `r` are integers, with `r` optional. It returns a kind type parameter that will provide at least `p` decimal digits of precision and a range of at least -10^r to 10^r, provided such a kind is available. It returns -1 if there is no kind with the requested range, -2 if there is no kind with the requested precision, and -3 if no kind with either the range or precision is available.

In the following example, s will be 16 for `sleet` which is different from that at the table of textbook.

```
integer, parameter :: s = selected_real_kind(20,50)
real(kind = s) :: x
x = 11.0
print *, "kind is: ", s
print *, "1/x is: ", 1.0/x
```

The output will be

```
kind is:  16
1/x is:  0.0909090909090909090909090909090909091
```

For integers, a similar function is:

`SELECTED_INT_KIND(r)`

where `r` is an integer. It returns a kind type parameter that will provide a range of at least -10^r to 10^r.

If we apply following statement in a program:

```
integer, parameter ::  R = selected_int_kind(20)
integer(kind=R) ::  f,n,i
```

Then an error message will display at the compile time:

```
ERROR: The kind type parameter value -1 is not valid for
type INTEGER.
```

A similar error appears, when try to set `selected_real_kind(35, 50)`.

```
program factorial
implicit none
integer, parameter :: R = selected_int_kind(18)
integer(kind=R) :: f,n,i
print *, "kind is: ", R

do n=14_R,19_R
f = 1_R
do i = 2_R, n
 f = f * i
end do
print *, n,"! = ", f
end do
end program factorial
```

In the program, `14_R` means that the integer is 14 with kind type `R`. The output of the above program is:

```
 kind is: 8
 14 ! =  87178291200
 15 ! =  1307674368000
 16 ! =  20922789888000
 17 ! =  355687428096000
 18 ! =  6402373705728000
 19 ! =  121645100408832000
```

```
program trykind
implicit none
integer, parameter :: s = selected_real_kind(25,50)
real(kind = s) :: x
real :: y
x = 11.0
y = 3.0
print *, "the real kind is: ", s
print *, "1/x is: ", 1.0/x
print *, "x*y is: ", x*y
print *, "y/x is: ", y/x
end program trykind
```

The output of the above program is:

```
 the real kind is:  16
 1/x is:  0.0909090909090909090909090909090909091
 x*y is:  33.0
 y/x is:  0.272727272727272727272727272727272727
```

The above example shows, when different types are mixed up together, the result is unexpectable.

It is important to ensure that all variables, arrays, and functions that are to have values of a particular kind are declared to be of the kind.

Another example:

```
integer, parameter :: s = selected_real_kind(25,50)
real(kind = s) ::  a
real :: y
y = 1/3.0
print *, "y is: ", y
a = y
print *, "a is: ", a
a = (y+3.7)**2
print *, "wrong number is: ", a
a = (1.0_s/3.0_s+3.7_s)**2
print *,"correct number is: ", a
```

The out put will be as follows:

```
y is:  0.33333334
a is:  0.3333333432674407958984375
wrong number is: 16.2677783966064453125
correct number is: 16.26777777777777777777777777777777
```

So mixed kind may cause errors.

The `COMPLEX` data type for a complex number $a + bi$.

Constant of a complex number in Fortran is: `(a,b)`. For example:
`(1.0, 1.0), (-6.0, 7.2), (-5.432, -1.4142)`

Declaration:

```
COMPLEX :: A, B
COMPLEX, DIMENSION(10, 10) :: Rho
INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(14)
COMPLEX(DP) :: Gamma
```

Intrinsic functions for complex numbers, where $z = a + bi$ is a complex number and x, y are real numbers:

`ABS(z)` returns $|z| = \sqrt{a^2 + b^2}$

`CONJ(z)` returns $\overline{z} = a - bi$

`EXP(z)` returns $e^z = e^a(\cos b + i \sin b)$

Some converting functions:

```
AIMAG(z)
CMPLX(x, y, KIND = k) or CMPLX(x, KIND = k)
REAL(z, KIND = k)
```

Input and output complex numbers using a pair of real numbers enclosed in parentheses.

The following calculates current in a circuit:

$$I = \frac{V}{Z},\ Z = R + \omega L i - \frac{i}{\omega C}$$

where R is resistance, L is the self-inductance, C is the capacitance and ω is the frequency.

```
REAL :: R, L, C, Omega
COMPLEX :: V, Z, I
... ...
Z = R + Omega*L*(0.0, 1.0) - (0.0,1.0) /(Omega * C)
I = V/Z
PRINT 10, I, ABS(I)
10 FORMAT(1X, "Current = ", F10.4, "+", &
  F10.4, "i" / 1x, "with magnitude = ", F10.4)
```

Structures in Fortran 90.

Suppose that we need to treat some records of students. Each records includes name, ID number, and marks. The normal array cannot store the data because all the elements in an array have the same data type.

Fortran uses derived data type to define user's data type.

```
TYPE Student_record
 CHARACTER(15) :: LastName, FirstName
 INTEGER :: IdNumber
 REAL(8) :: Marks
END TYPE Student_record
```

In general, the derived type definition is

```
TYPE type-name
 declare1
 declare2
 ...
END TYPE type-name
```

Each data item in a structure is called a component. After a type is defined, it can be declared for variables or arrays.

```
TYPE(type-name) :: A
TYPE(type-name), DIMENSION(20) :: B
```

A values of a derived type variable is of the form `type-name(list of component values)`.

```
TYPE(Student_record), DIMENSION(60) :: records
TYPE(Student_record) :: sample
sample = Student_record("John","Smith",12345, &
       (/ 90, 95, 87, 35, 67, 90, 89, 80 /))
records(1) = sample
```

Individual component of a structure can be accessed using component selector (%).

```
structure-name%component-name
```

Example:

```
records(2)%LastName = "Anderson"

READ *, records(3)%IdNumber

PRINT *, sample%Marks
```

The following example defines a structure of type `Point`.

```
TYPE Point
  REAL :: X, Y
END TYPE

TYPE(Point) :: P, Q, R
REAL :: A = 2.0
P = Point(2.5, 3.2)
Q = Point(A, 2*A)
R%X = Q%X
R%Y = P%Y
```

Another example see p 242.