FORTRAN - NUMBERS

Numbers in Fortran are represented by three intrinsic data types:

- Integer type
- Real type
- Complex type

Integer Type

The integer types can hold only integer values. The following example extracts the largest value that could be hold in a usual four byte integer:

```
program testingInt
implicit none

   integer :: largeval
   print *, huge(largeval)

end program testingInt
```

When you compile and execute the above program it produces the following result:

```
2147483647
```

Please note that the **huge** function gives the largest number that can be held by the specific integer data type. You can also specify the number of bytes using the **kind** specifier. The following example demonstrates this:

```
program testingInt
implicit none

   !two byte integer
   integer(kind=2) :: shortval

   !four byte integer
   integer(kind=4) :: longval

   !eight byte integer
   integer(kind=8) :: verylongval

   !sixteen byte integer
   integer(kind=16) :: veryverylongval

   !default integer
   integer :: defval

   print *, huge(shortval)
   print *, huge(longval)
   print *, huge(verylongval)
   print *, huge(veryverylongval)
   print *, huge(defval)

end program testingInt
```

When you compile and execute the above program it produces the following result:

```
32767
2147483647
9223372036854775807
170141183460469231731687303715884105727
```

```
2147483647
```

Real Type

It stores the floating point numbers, such as 2.0, 3.1415, -100.876, etc.

Traditionally there were two different **real** types : the default real type and **double precision** type.

However, Fortran 90/95 provides more control over the precision of real and integer data types through the **kind** specifier, which we will study shortly.

The following example shows the use of real data type:

```
program division
implicit none

   ! Define real variables
   real :: p, q, realRes

   ! Define integer variables
   integer :: i, j, intRes

   ! Assigning  values
   p = 2.0
   q = 3.0
   i = 2
   j = 3

   ! floating point division
   realRes = p/q
   intRes = i/j

   print *, realRes
   print *, intRes

end program division
```

When you compile and execute the above program it produces the following result:

```
0.666666687
0
```

Complex Type

This is used for storing complex numbers. A complex number has two parts : the real part and the imaginary part. Two consecutive numeric storage units store these two parts.

For example, the complex number $3.0, -5.0$ is equal to 3.0 – 5.0i

The generic function **cmplx** creates a complex number. It produces a result who's real and imaginary parts are single precision, irrespective of the type of the input arguments.

```
program createComplex
implicit none

   integer :: i = 10
   real :: x = 5.17
   print *, cmplx(i, x)

end program createComplex
```

When you compile and execute the above program it produces the following result:

```
(10.0000000, 5.17000008)
```

The following program demonstrates complex number arithmetic:

```
program ComplexArithmatic
implicit none

   complex, parameter :: i = (0, 1)   ! sqrt(-1)
   complex :: x, y, z

   x = (7, 8);
   y = (5, -7)
   write(*,*) i * x * y

   z = x + y
   print *, "z = x + y = ", z

   z = x - y
   print *, "z = x - y = ", z

   z = x * y
   print *, "z = x * y = ", z

   z = x / y
   print *, "z = x / y = ", z

end program ComplexArithmatic
```

When you compile and execute the above program it produces the following result:

```
(9.00000000, 91.0000000)
z = x + y = (12.0000000, 1.00000000)
z = x - y = (2.00000000, 15.0000000)
z = x * y = (91.0000000, -9.00000000)
z = x / y = (-0.283783793, 1.20270276)
```

The Range, Precision and Size of Numbers

The range on integer numbers, the precision and the size of floating point numbers depends on the number of bits allocated to the specific data type.

The following table displays the number of bits and range for integers:

Number of bits	Maximum value	Reason
64	9,223,372,036,854,774,807	2 * * 63–1
32	2,147,483,647	2 * * 31–1

The following table displays the number of bits, smallest and largest value, and the precision for real numbers.

Number of bits	Largest value	Smallest value	Precision
64	0.8E+308	0.5E–308	15–18
32	1.7E+38	0.3E–38	6-9

The following examples demonstrate this:

```
program rangePrecision
implicit none
```

```
   real:: x, y, z
   x = 1.5e+40
   y = 3.73e+40
   z = x * y
   print *, z

end program rangePrecision
```

When you compile and execute the above program it produces the following result:

```
x = 1.5e+40
    1
Error : Real constant overflows its kind at (1)
main.f95:5.12:

y = 3.73e+40
    1
Error : Real constant overflows its kind at (1)
```

Now let us use a smaller number:

```
program rangePrecision
implicit none

   real:: x, y, z
   x = 1.5e+20
   y = 3.73e+20
   z = x * y
   print *, z

   z = x/y
   print *, z

end program rangePrecision
```

When you compile and execute the above program it produces the following result:

```
Infinity
0.402144760
```

Now let's watch underflow:

```
program rangePrecision
implicit none

   real:: x, y, z
   x = 1.5e-30
   y = 3.73e-60
   z = x * y
   print *, z

   z = x/y
   print *, z

end program rangePrecision
```

When you compile and execute the above program it produces the following result:

```
y = 3.73e-60
    1
Warning : Real constant underflows its kind at (1)

Executing the program....
$demo

0.00000000E+00
```

```
Infinity
```

The Kind Specifier

In scientific programming, one often needs to know the range and precision of data of the hardware platform on which the work is being done.

The intrinsic function **kind** allows you to query the details of the hardware's data representations before running a program.

```
program kindCheck
implicit none

   integer :: i
   real :: r
   complex :: cp
   print *,' Integer ', kind(i)
   print *,' Real ', kind(r)
   print *,' Complex ', kind(cp)

end program kindCheck
```

When you compile and execute the above program it produces the following result:

```
Integer 4
Real 4
Complex 4
```

You can also check the kind of all data types:

```
program checkKind
implicit none

   integer :: i
   real :: r
   character*1 :: c
   logical :: lg
   complex :: cp

   print *,' Integer ', kind(i)
   print *,' Real ', kind(r)
   print *,' Complex ', kind(cp)
   print *,' Character ', kind(c)
   print *,' Logical ', kind(lg)

end program checkKind
```

When you compile and execute the above program it produces the following result:

```
Integer 4
Real 4
Complex 4
Character 1
Logical 4
```