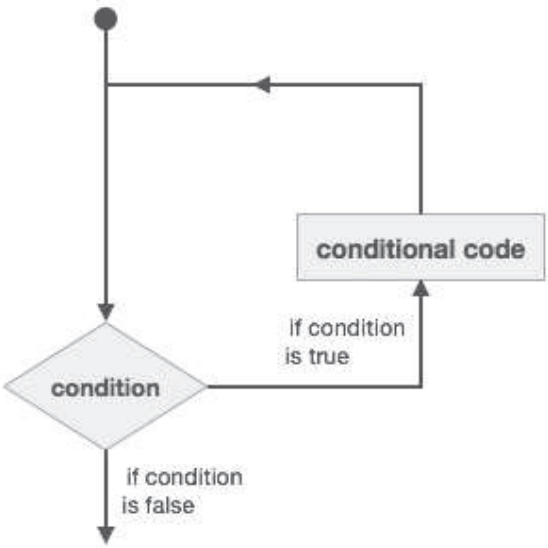
9. Fortran – Loops

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially : The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Fortran provides the following types of loop constructs to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
do loop	This construct enables a statement, or a series of statements, to be carried out iteratively, while a given condition is true.
do while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
nested loops	You can use one or more loop construct inside any other loop construct.

do Loop

The do loop construct enables a statement, or a series of statements, to be carried out iteratively, while a given condition is true.

Syntax

The general form of the do loop is:

```
do var = start, stop [,step]
   ! statement(s)
   …
end do
```

Where,

- the loop variable var should be an integer
- start is initial value
- stop is the final value
- step is the increment, if this is omitted, then the variable var is increased by unity

For example:

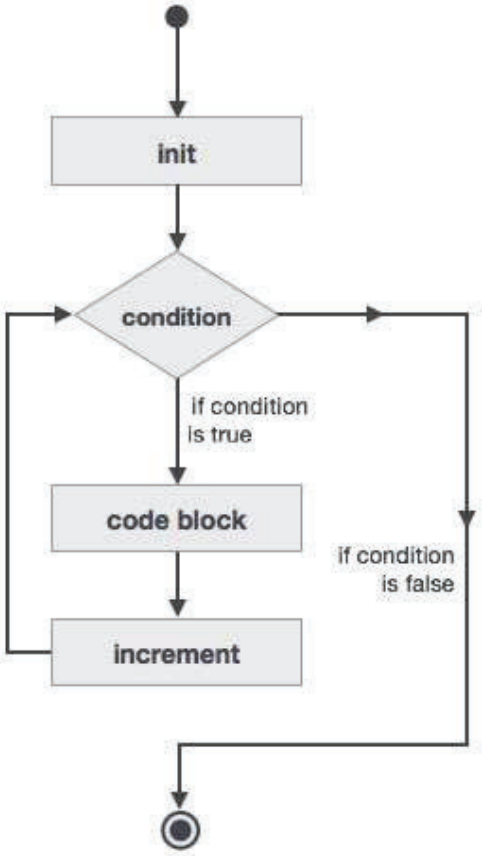
```
! compute factorials
do n = 1, 10
   nfact = nfact * n
   ! printing the value of n and its factorial
   print*,  n, " ", nfact
end do
```

Flow Diagram

Here is the flow of control for the do loop construct:

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables. In our case, the variable var is initialised with the value start.
- Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the loop. In our case, the condition is that the variable var reaches its final value stop.

- After the body of the loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update the loop control variable var.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the loop terminates.



Example 1

This example prints the numbers 11 to 20:

```
program printNum
implicit none

   ! define variables
   integer :: n

   do n = 11, 20
      ! printing the value of n
```

```
      print*,  n
   end do

end program printNum
```

When the above code is compiled and executed, it produces the following result:

```
11
12
13
14
15
16
17
18
19
20
```

Example 2

This program calculates the factorials of numbers 1 to 10:

```
program factorial
implicit none

   ! define variables
   integer :: nfact = 1
   integer :: n

   ! compute factorials
   do n = 1, 10
      nfact = nfact * n
      ! print values
      print*,  n, " ", nfact
   end do

end program factorial
```

When the above code is compiled and executed, it produces the following result:

```
1          1
2          2
3          6
4         24
5        120
6        720
7       5040
8      40320
9     362880
10   3628800
```

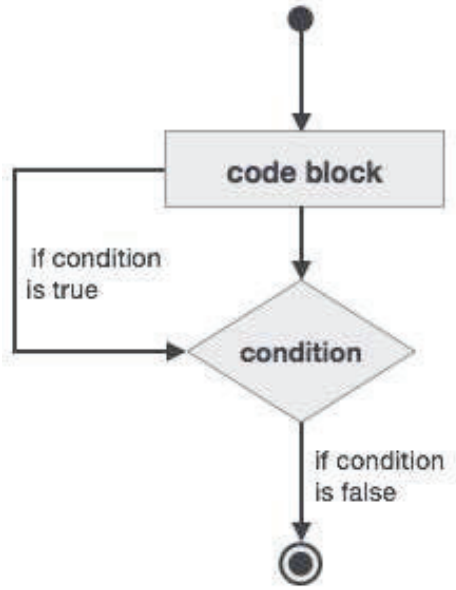
do-while Loop

It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.

Syntax

```
do while (logical expr)
   statements
end do
```

Flow Diagram



Example

```
program factorial
implicit none

   ! define variables
   integer :: nfact = 1
   integer :: n = 1

   ! compute factorials
   do while (n <= 10)
      nfact = nfact * n
      n = n + 1
      print*,  n, " ", nfact
   end do
end program factorial
```

When the above code is compiled and executed, it produces the following result:

```
2          1
3          2
4          6
5         24
6        120
7        720
8       5040
9      40320
10   362880
11  3628800
```

Nested Loops

You can use one or more loop construct inside any another loop construct. You can also put labels on loops.

Syntax

```
iloop: do i = 1, 3
   print*, "i: ", i
      jloop: do j = 1, 3
      print*, "j: ", j

      kloop: do k = 1, 3
         print*, "k: ", k

      end do kloop
   end do jloop
end do iloop
```

Example

```
program nestedLoop
implicit none

   integer:: i, j, k

   iloop: do i = 1, 3
      jloop: do j = 1, 3
         kloop: do k = 1, 3

            print*, "(i, j, k): ", i, j, k

         end do kloop
      end do jloop
   end do iloop

end program nestedLoop
```

When the above code is compiled and executed, it produces the following result:

```
(i, j, k): 1  1  1
(i, j, k): 1  1  2
(i, j, k): 1  1  3
(i, j, k): 1  2  1
(i, j, k): 1  2  2
(i, j, k): 1  2  3
(i, j, k): 1  3  1
(i, j, k): 1  3  2
(i, j, k): 1  3  3
(i, j, k): 2  1  1
(i, j, k): 2  1  2
(i, j, k): 2  1  3
(i, j, k): 2  2  1
(i, j, k): 2  2  2
(i, j, k): 2  2  3
(i, j, k): 2  3  1
(i, j, k): 2  3  2
(i, j, k): 2  3  3
(i, j, k): 3  1  1
(i, j, k): 3  1  2
(i, j, k): 3  1  3
(i, j, k): 3  2  1
(i, j, k): 3  2  2
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Fortran supports the following control statements. Click the following links to check their detail.

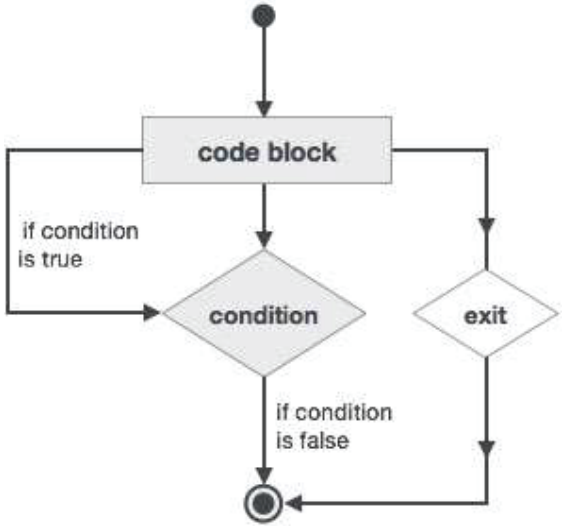
Control Statement	Description
exit	If the exit statement is executed, the loop is exited, and the execution of the program continues at the first executable statement

	after the end do statement.
cycle	If a cycle statement is executed, the program continues at the start of the next iteration.
stop	If you wish execution of your program to stop, you can insert a stop statement

Exit Statement

Exit statement terminates the loop or select case statement, and transfers execution to the statement immediately following the loop or select.

Flow Diagram



Example

```
program nestedLoop
implicit none

integer:: i, j, k
   iloop: do i = 1, 3
      jloop: do j = 1, 3
         kloop: do k = 1, 3

         print*, "(i, j, k): ", i, j, k
```

```
         if (k==2) then
            exit jloop
         end if
         end do kloop
      end do jloop
   end do iloop
end program nestedLoop
```

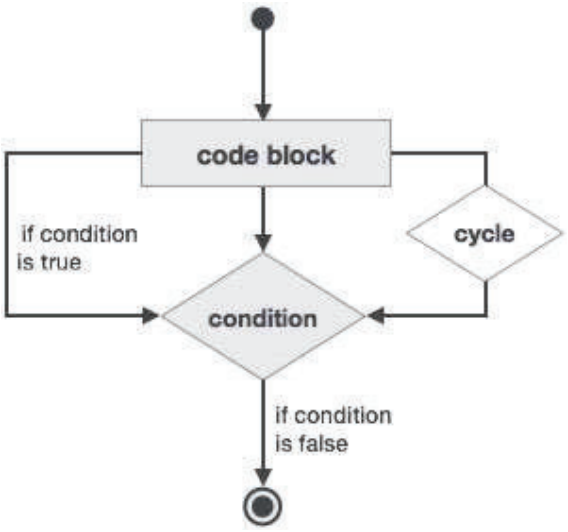
When the above code is compiled and executed, it produces the following result:

```
(i, j, k): 1  1  1
(i, j, k): 1  1  2
(i, j, k): 2  1  1
(i, j, k): 2  1  2
(i, j, k): 3  1  1
(i, j, k): 3  1  2
```

Cycle Statement

The cycle statement causes the loop to skip the remainder of its body, and immediately retest its condition prior to reiterating.

Flow diagram



Example

```
program cycle_example
implicit none

   integer :: i

   do i = 1, 20

      if (i == 5) then
         cycle
      end if
   print*, i
   end do
end program cycle_example
```

When the above code is compiled and executed, it produces the following result:

```
1
2
3
4
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Stop Statement

If you wish execution of your program to cease, you can insert a stop statement.

Example

```
program stop_example
implicit none

   integer :: i
   do i = 1, 20

      if (i == 5) then
         stop
      end if

      print*, i
   end do

end program stop_example
```

When the above code is compiled and executed, it produces the following result:

```
1
2
3
4
```

10. 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
```