FORTRAN LESSON 3

In Fortran Lesson 1 we briefly looked at the types of variables in Fortran. To avoid mistakes in Fortran arithmetic you must pay close attention to rules regarding working with numbers of the various types. Whereas Basic is more lenient, allowing some flexibility in mixing variables and numbers of different types, Fortran is less forgiving and will make you pay for oversights. In this lesson we look more closely at some of the rules and conventions that must be observed.

Integers

An *integer* in Fortran is a whole number; it cannot contain commas or a decimal point. Examples of numbers considered integers by Fortran are

12 , -1311 , 0 , +43 , 123456789 .

For positive integers the plus sign is optional, but negative integers must be preceded by a minus sign. Examples of numbers *not* considered integers by Fortran are

22,547 , 3. , 4.0 , -43.57 .

Because of the decimal points, Fortran will regard 3. and 4.0 as *real* numbers.

An integer N in GNU Fortran must lie within the range

$$- 2,147,483,648 \leq N \leq 2,147,483,647 \ .$$

One idiosyncrasy of Fortran is that when it performs arithmetic on integers, it insists on giving an answer that is likewise an integer. If the answer is not really an integer, Fortran makes it one by discarding the decimal point and all digits thereafter. For example, Fortran will assert that

$$11/8 = 1 \quad , \quad 15/4 = 3 \quad , \quad -4/3 = -1 \quad , \quad -50/6 = -8 \quad , \quad 2/3 = 0 \quad .$$

If you want Fortran to give you the correct value of 11/8, you tell it to compute 11./8., so that it interprets the numbers as real numbers and produces the correct value 1.375. Integer arithmetic in Fortran can lead to other weird surprises - for instance, the distributive law of division is invalid, as demonstrated by the example

(2 + 3)/4 = 5/4 = 1   but   (2/4) + (3/4) = 0 + 0 = 0   .

Most of the built-in functions in Fortran apply to real numbers, and attempts to apply them to integers result in compiler error messages. The compiler will protest if you ask Fortran to compute sqrt(5), but it has no problem with sqrt(5.). Likewise, if you declare N to be an integer variable and ask Fortran to compute sqrt(N) or cos(N) or log(N), your program will not compile since these functions cannot act on integers. One way around this problem is to use the intermediate function

real(x)   ,

which converts x to a real number (if it is not already one). Then, for example,

real(5) = 5.   ,   sqrt(real(5)) = sqrt(5.) = 2.23606801 .

The compiler will have no objection if N is an integer variable and you ask Fortran to compute a composition like sqrt(real(N)) or cos(real(N)).

If you declare that A is an integer and later make the assignment A = 3.45, Fortran will not complain but it will truncate 3.45 and assign A the value A = 3. Likewise, if you insert the statement A = sqrt (5.), Fortran will truncate sqrt (5.) = 2.23606801 and deduce that A = 2. But errors such as these are easily avoided if you are careful to make correct type declaration statements for all variables at the beginning of your program.

Single Precision Real Numbers

A *real number*, or more precisely a *single precision real number*, is written with a decimal point by Fortran, even when it is a whole number. The sequence of statements

```
real x
integer y
x = 3
y = 3
print *, "x = ", x, " but y = ", y, "  -  weird!"
```

produces the output

```
x = 3. but y = 3  -  weird!
```

GNU Fortran uses up to 9 digits, not counting the decimal point, to represent real numbers. It will report that

sqrt (3.) = 1.73205078   ,   sqrt (1100.) = 33.1662483   ,   sqrt (2.25) = 1.5   .

Fortran can use also *scientific notation* to represent real numbers. The sequence "En" attached to the end of a number, where n is an integer, means that the number is to be multiplied by 10^n. Here are various ways of writing the number 12.345:

1.2345E1 , .12345E2 , .012345E3 , 12.345E0 , 12345E-3 .

In working in single precision it is futile to assign more than 9 or 10 nonzero digits to represent a number, as Fortran will change all further digits to 0. (The 10th digit can affect how Fortran does the truncation.)

The assignments

x = 123456789876543. , x = 123456789800000. , x = 1234567898E5

produce the same result if x already has been declared a single precision real number. Note that commas are not used in representing numbers; as helpful as they might be to humans, computers find them unnecessary.

Double Precision Real Numbers

A *double precision real number* in GNU Fortran can be represented by up to 17 digits before truncation occurs. Double precision numbers are written in scientific notation but with D usurping the role of E. Some various ways of writing the number 12.345 as a double precision real number are

1.2345D1 , .12345D2 , .012345D3 , 12.345D0 , 12345D-3 .

When assigning a value to a double precision variable you should use this D-scientific notation, as otherwise the value will be read only in single precision. For example, if A is double precision and you want to assign A the value 3.2, you should write

A = 3.2D0

instead of just A = 3.2. (See *Base 2 Conversion Errors* below for more explanation.)

When a number is input from the keyboard in response to a "read *" command, the user need not worry about types or input format. Suppose for example that x is single or double precision, and the user is to enter a value for x in response to the command "read *, x". If the user enters simply "3" (integer format), GNU Fortran will change 3 to the proper format (to 3. if x is single precision and to 3D0 if x is double precision) before assigning it to x. Likewise, if x is double precision and the user enters 3.1 (single precision format), Fortran converts 3.1 to 3.1D0 before assigning it to x. (However, with an ordinary assignment statement "x = 3.1" from within the program, the number is *not* changed to double precision format before being assigned to x.)

A number x can be converted to double precision by the function

dble(x)   .

Base 2 Conversion Errors

Whereas humans, having 10 fingers, do arithmetic in base 10, computers have no fingers but do arithmetic with on-off switches and therefore use base 2. As we know, some numbers have infinite decimal representations in base 10, such as

1/3 = .33333 …   ,   2/7 = .285714285714 …   .

There is no way to represent such numbers in base 10 with a finite number of digits without making a round-off error. Computers have the same problem working in base 2. In general, the only numbers representable with a finite number of digits in base 2 can be written in the form m/n, where m and n are integers and n is an integral power of 2. Examples are

$6\ (= 6/2^0)$, 5/2 , 3/8 , 29/16 , 537/256 , -3/1024 .

When we ask computers to do arithmetic for us, there is an inevitable source of error. We give the computer the numbers in base 10, and the computer must change them all over to base 2. For most numbers there is a round-off error, as the computer can work with only a finite number of digits at a time, and most numbers do not have a finite representation in base 2. If the computer is working in single precision Fortran, it works in about 9 digits (base 10), and so the round-off error will occur in about the 8th or 9th base 10 digit. In double precision this error appears much later, in about the 16th or 17th base 10 digit. If the arithmetic the computer performs is very complicated, these round-off errors can accumulate on top of each other until the total error in the end result is much larger. After the computer has done its job in base 2, it converts all numbers back to base 10 and reports its results.

Even if the computer does no arithmetic at all, but just prints out the numbers, the base 2 conversion error still appears. Here is a program illustrating the phenomenon:

```
program demo
real x
double precision y, z
x = 1.1
y = 1.1
z = 1.1D0
print *, "x =", x, " , y =", y, " , z =", z
end
```

The somewhat surprising output when this program is run in GNU Fortran is

```
x = 1.10000002   ,   y = 1.10000002   ,   z = 1.1   .
```

The variable x is single precision, and base 2 conversion round-off error shows up in the 9th digit. Although y is double precision, it has the same round-off error as x because the value 1.1 is assigned to y only in single precision mode. (What happens is Fortran converts 1.1 to base 2 *before* changing it to double precision and assigning it to y.) Since z is double precision, and it is assigned the value 1.1 in double precision mode, round-off error occurs much later, far beyond the nine digits in which the results are printed. Thus the value of z prints exactly as it is received. Using *write* and *format* statements (see below), it is possible to print z using 17 digits; if you do so, you will find that Fortran reports z = 1.1000000000000001, where the final erroneous 1 appears as the 17th digit.

Base 2 round-off error occurs in the preceding example because 1.1 = 11/10, and 10 is not a power of 2. If you modify the program by replacing 1.1 with 1.125 = 9/8, there will be no round-off error because $8 = 2^3$ is a power of 2 - so the values of x, y, and z will print exactly as assigned. (Try it!!)

Mixed Type Arithmetic

In general, arithmetic in Fortran that mixes numbers of different types should be avoided, as the rules are quickly forgotten and mistakes are easily made. If Fortran is asked in some arithmetic operation to combine an integer number with a real one, usually it will wait until it is forced to combine the two and then convert the integer to real mode. Here are some calculations illustrating the process followed by Fortran, and showing why you should stay away from this nonsense:

```
5. * (3 / 4) = 5. * 0 = 5. * 0. = 0.
(5. * 3) / 4 = (5. * 3.) / 4 = 15. / 4 = 15. / 4. = 3.75
5. + 3 / 4 = 5. + 0 = 5. + 0. = 5.
5 + 3. / 4 = 5 + 3. / 4. = 5 + .75 = 5. + .75 = 5.75
```

If x and y are declared as double precision variables, and you want to multiply x by a number, say 2.1 for example, to get y, you should write

```
y = 2.1D0 * x   .
```

Writing just y = 2.1 * x will retain single precision when 2.1 is converted to base 2, thereby introducing a larger base 2 round-off error and defeating your efforts at double precision. Similar remarks apply to other arithmetic operations. Errors of this nature are easily made when working in double precision. The best way to avoid them is to follow religiously this general rule:

Do not mix numbers of different types in Fortran arithmetic!!

Exponentials and Roots

Already we point out an exception to the above rule - it is OK to use integers as exponents of real numbers. That is because, when serving as an exponent, an integer acts more as a "counter of multiplications" rather than as an active participant in the arithmetic. For instance, when Fortran does the calculation 1.2^5, it performs the multiplications

```
1.2 * 1.2 * 1.2 *1.2 * 1.2   ,
```

and the integer 5 never enters into the calculations! Thus, although it may appear so at first glance, the computation of 1.2^5 does not really mix an integer with a real number in any arithmetic operation. The same can be said of negative integers as exponents. The calculation of 1.2^{-5} involves multiplying five factors of 1.2, and then taking the reciprocal of the result - so the number -5 is not involved in the actual arithmetic.

Rational exponents must be handled carefully. A common mistake of novice Fortran programmers is to write something like 5 ** (2/3) and expect Fortran to compute the value of $5^{2/3}$. But Fortran will view 2 and 3 as integers and compute 2/3 = 0, and conclude that 5 ** (2/3) = 5 ** 0 = 1. The correct expression for computing $5^{2/3}$ is

```
5. ** (2./3.)   ,
```

wherein all numbers are viewed as real numbers.

Roots of numbers are computed in the same manner. To compute the seventh root of 3 you would use the expression

```
3. ** (1./7.)   .
```

If N is an integer variable and you wish to compute the N-th root of the real variable x, do not write x ** (1/N), as Fortran will interpret 1/N as 0 when N > 1. Instead write x ** (1./real (N)), so that 1 and N are first converted to real variables.

Write and Format Statements

Just as in Basic we use TAB and PRINT USING commands to more precisely control program output, in Fortran we can use *write* commands with *format* statements. While these can get complicated, the most commonly used options are pretty easy to use. A typical *write* statement is

```
write (*,20) x, y, z    .
```

The "*" in the parentheses instructs Fortran to write to the screen, while "20" refers to the label of the *format* statement for this *write* command. The x, y, and z are the variables to be printed. A *format* statement for this *write* command might be

```
20      format (3f10.4)   .
```

Inside the parentheses, the "3" indicates that 3 entities will be printed, the "f" denotes that these will be *floating point* real numbers (not exponential notation), the "10" stipulates that 10 places will be used for printing (counting the sign, decimal point, and the digits), and ".4" mandates 4 digits after the decimal point. Some printouts formatted this way are

```
12345.6789   ,   -1234.5678   ,   10002.3400   .
```

The letter "f" in this context is a *format code letter*; here are some of the more commonly used format code letters, with their implications:

f	real number, floating point format
e	single precision real number, exponential notation
d	double precision real number, exponential notation
i	integer
a	text string (character)
x	space
/	vertical space (line feed)
t	tab indicator

Strings (in quotes) may be placed in format statements, separated by commas. Here are examples of write statements with corresponding format statements; at the right of each is a description of the corresponding output:

write (*,10) n, x, y 10 format (i4,4x,f10.4,2x,f10.4)	integer n printed using 4 places, then 4 spaces, then real numbers x and y printed with 2 spaces between, each using 10 places and 4 decimal places
write (*,20) area 20 format ("The area is ",f8.5)	string in quotes is printed, then the real number area is printed, using 8 places with 5 decimal places
write (*,30) "The area is ", area 30 format (a,f8.5)	same output as immediately above
write (*,40) x, y, z 40 format (3d20.14)	3 double precision numbers x, y, z printed, each reserving 20 spaces, with 14 decimal places
write (*,50) student, score 50 format (a20,4x,i3)	student, a text string up to 20 characters, is printed, then 4 spaces, then score, an integer using a maximum of 3 places
write (*,60) r, A	tabs to column 10, prints real

60 format (t10,f4.2,/,t10,f6.2)	number r, goes to next line, tabs to column 10, prints real number A

You can use loops with format statements to print arrays; here are examples:

```
     do i = 1, 10
       write (*,70) a(i)
     end do
70   format (f5.2)
```
	an array a of real numbers, indexed from 1 to 10, is printed; each entry occupies 5 places with 2 decimal places, and is printed on a separate line

```
     write (*,80) (a(i), i = 1, 10)
80   format (f5.2)
```
	same output as immediately above

```
     write (*,90) (a(i), i = 1, 10)
90   format (10f5.2)
```
	same output as above, except that all entries are printed on the same line

```
     do i = 1, 5
        write (*,7) (m(i,j), j = 1, 6)
7       format (6i3)
     end do
```
	prints a 5 x 6 two-dimensional array m of integers, with each integer entry m(i,j) occupying 3 places. Each row of the matrix appears on its own line.

Here are other useful things to know about formatting:

1. If you do not specify a format, GNU Fortran will print real numbers using about 9 digits, even if you do calculations in double precision. If you want to print in double precision you must use write and format statements. When double precision is used the maximum number of digits possible is 17. A format specifier something like format (fm.n), where m is at least 20, is required to take full advantage of double precision.

2. If a value is too large to be printed in the specified format, Fortran will just print a string of asterisks (eg: **********). If you get such an output, you have to fix your format statement.

3. Real numbers are rounded off (not truncated) to fit the specified formatting.

4. If your formatting specifies more positions than the number requires, blanks are inserted to the left of the number.

5. Format statements may appear anywhere in a program after the variable declarations and before the *end* statement.

6. Unless your format statement is very simple, the chances are that your output won't look like you want on the first try - just fiddle with the formatting until you get it right.

Following are examples of stored values, formatting specifications for printing the values, and resulting output. (The "^" symbol indicates a blank).

Stored Value	**Format Specifier**	**Output**
1.234567	f8.2	^^^^1.23
0.00001	f5.3	0.000

-12345	i5	*****
-12345	i6	-12345
12345	i6	^12345
0.00001234	e10.3	^0.123E-04
0.0001234	e12.4	^^0.1234E-03
1234567.89	e9.2	^0.12E+07
aloha	a8	^^^aloha
1.23456789123D0	d17.10	^0.1234567891E+01