

fortran 77 tutorial

(nearly identical versions available at several sites online)

1. Preface

The goal of this Fortran tutorial is to give a quick introduction to the most common features of the Fortran 77 programming language. It is not a complete reference! Many details have been omitted. The presentation focuses on scientific computations, mainly linear algebra. The outline of this tutorial was inspired by the excellent book "Handbook for Matrix Computations" by T.F. Coleman and C. Van Loan, published by SIAM (unfortunately this book is out of print).

This tutorial was designed to be used in the course SCCM-001-F: Introduction to Fortran at Stanford University, 1996. Permission to use this tutorial for educational and other non-commercial purposes is granted provided all author and copyright information is retained.

Erik Boman, Stanford, December 1995.

2. What is Fortran?

Fortran is a general purpose programming language, mainly intended for mathematical computations in e.g. engineering. Fortran is an acronym for FORMula TRANslation, and was originally capitalized as FORTRAN. However, following the current trend to only capitalize the first letter in acronyms, we will call it Fortran. Fortran was the first ever high-level programming languages. The work on Fortran started in the 1950's at IBM and there have been many versions since. By convention, a Fortran version is denoted by the last two digits of the year the standard was proposed. Thus we have

- * Fortran 66
- * Fortran 77
- * Fortran 90 (95)

The most common Fortran version today is still Fortran 77, although Fortran 90 is growing in popularity. Fortran 95 is a revised version of Fortran 90 which is expected to be approved by ANSI soon (1996). There are also several versions of Fortran aimed at parallel computers. The most important one is High Performance Fortran (HPF), which is a de-facto standard.

Users should be aware that most Fortran 77 compilers allow a superset of Fortran 77, i.e. they allow non-standard extensions. In this tutorial we will emphasize standard ANSI Fortran 77.

Why learn Fortran?

Fortran is the dominant programming language used in engineering applications. It is therefore important for engineering graduates to be able to read and modify Fortran code. From time to time, so-called experts predict that Fortran will rapidly fade in popularity and soon become extinct. These predictions have always failed. Fortran is

the most enduring computer programming language in history. One of the main reasons Fortran has survived and will survive is software inertia. Once a company has spent many man-years and perhaps millions of dollars on a software product, it is unlikely to try to translate the software to a different language. Reliable software translation is a very difficult task.

Portability

A major advantage Fortran has is that it is standardized by ANSI and ISO (see footnotes). Consequently, if your program is written in ANSI Fortran 77 then it will run on any computer that has a Fortran 77 compiler. Thus, Fortran programs are portable across machine platforms. (If you want to read some Fortran Standards Documents, [click here](#).)

3. Fortran 77 Basics

A Fortran program is just a sequence of lines of text. The text has to follow a certain syntax to be a valid Fortran program. We start by looking at a simple example:

```
program circle
real r, area
```

c This program reads a real number r and prints
c the area of a circle with radius r.

```
write (*,*) 'Give radius r:'
read (*,*) r
area = 3.14159*r*r
write (*,*) 'Area = ', area

stop
end
```

The lines that begin with with a "c" are comments and has no purpose other than to make the program more readable for humans. Originally, all Fortran programs had to be written in all upper-case letters. Most people now write lower-case since this is more legible, and so will we.

Program organization

A Fortran program generally consists of a main program (or driver) and possibly several subprograms (or procedures or subroutines). For now we will assume all the statements are in the main program; subprograms will be treated later. The structure of a main program is:

```
program name

declarations

statements
```

```
stop
end
```

In this tutorial, words that are in italics should not be taken as literal text, but rather as a generic description. The stop statement is optional and may seem superfluous since the program will stop when it reaches the end anyways, but it is recommended to always terminate a program with the stop statement to emphasize that the execution flow stops there.

Column position rules

Fortran 77 is not a free-format language, but has a very strict set of rules for how the source code should be formatted. The most important rules are the column position rules:

- Col. 1 : Blank, or a "c" or "*" for comments
- Col. 2-5 : Statement label (optional)
- Col. 6 : Continuation of previous line (optional)
- Col. 7-72 : Statements
- Col. 73-80: Sequence number (optional, rarely used today)

Most lines in a Fortran 77 program starts with 6 blanks and ends before column 72, i.e. only the statement field is used. Note that Fortran 90 allows free format.

Comments

A line that begins with the letter "c" or an asterisk in the first column is a comment. Comments may appear anywhere in the program. Well-written comments are crucial to program readability. Commercial Fortran codes often contain about 50% comments. You may also encounter Fortran programs that use the exclamation mark (!) for comments. This is highly non-standard in Fortran 77, but is allowed in Fortran 90. The exclamation mark may appear anywhere on a line (except in positions 2-6).

Continuation

Occasionally, a statement does not fit into one single line. One can then break the statement into two or more lines, and use the continuation mark in position 6. Example:

```
c23456789 (This demonstrates column position!)
```

```
c The next statement goes over two physical lines
  area = 3.14159265358979
+      * r * r
```

Any character can be used instead of the plus sign as a continuation character. It is considered good programming style to use either the plus sign, an ampersand, or numbers (2 for the second line, 3 for the third, and so on).

Blank spaces

Blank spaces are ignored in Fortran 77. So if you remove all blanks in a Fortran 77 program, the program is still syntactically correct but almost unreadable for humans.

5. Variables, types, and declarations

Variable names

Variable names in Fortran consist of 1-6 characters chosen from the letters a-z and the digits 0-9. The first character must be a letter. (Note: Fortran 90 allows variable names of arbitrary length). Fortran 77 does not distinguish between upper and lower case, in fact, it assumes all input is upper case. However, nearly all Fortran 77 compilers will accept lower case. If you should ever encounter a Fortran 77 compiler that insists on upper case it is usually easy to convert the source code to all upper case.

Types and declarations

Every variable should be defined in a declaration. This establishes the type of the variable. The most common declarations are:

```
integer list of variables
real list of variables
double precision list of variables
complex list of variables
logical list of variables
character list of variables
```

The list of variables should consist of variable names separated by commas. Each variable should be declared exactly once. If a variable is undeclared, Fortran 77 uses a set of implicit rules to establish the type. This means all variables starting with the letters i-n are integers and all others are real. Many old Fortran 77 programs use these implicit rules, but you should not! The probability of errors in your program grows dramatically if you do not consistently declare your variables.

Integers and floating point variables

Fortran 77 has only one type for integer variables. Integers are usually stored as 32 bits (4 bytes) variables. Therefore, all integer variables should take on values in the range $[-m, m]$ where m is approximately $2 \cdot 10^9$.

Fortran 77 has two different types for floating point variables, called real and double precision. While real is often adequate, some numerical calculations need very high precision and double precision should be used. Usually a real is a 4 byte variable and the double precision is 8 bytes, but this is machine dependent. Some non-standard Fortran versions use the syntax `real*8` to denote 8 byte floating point variables.

The parameter statement

Some constants appear many times in a program. It is then often desirable to define them only once, in the beginning of the program. This is what the parameter statement is for. It also makes programs more readable. For example, the circle area program should rather have been written like this:

```
program circle
real r, area, pi
parameter (pi = 3.14159)
```

c This program reads a real number r and prints

c the area of a circle with radius r.

```
write (*,*) 'Give radius r:'  
read (*,*) r  
area = pi*r*r  
write (*,*) 'Area = ', area  
  
stop  
end
```

The syntax of the parameter statement is

```
parameter (name = constant, ... , name = constant)
```

The rules for the parameter statement are:

- * The "variable" defined in the parameter statement is not a variable but rather a constant whose value can never change
- * A "variable" can appear in at most one parameter statement
- * The parameter statement(s) must come before the first executable statement

Some good reasons to use the parameter statement are:

- * it helps reduce the number of typos
- * it is easy to change a constant that appears many times in a program

6. Expressions and assignment

Constants

The simplest form of an expression is a constant. There are 6 types of constants, corresponding to the 6 data types. Here are some integer constants:

```
1  
0  
-100  
32767  
+15
```

Then we have real constants:

```
1.0  
-0.25  
2.0E6  
3.333E-1
```

The E-notation means that you should multiply the constant by 10 raised to the power following the "E". Hence, 2.0E6 is two million, while 3.333E-1 is approximately one third.

For constants that are larger than the largest real allowed, or that requires high precision, double precision should be used. The notation is the same as for real constants except the "E" is replaced by a "D". Examples:

```
2.0D-1
1D99
```

Here 2.0D-1 is a double precision one-fifth, while 1D99 is a one followed by 99 zeros.

The next type is complex constants. This is designated by a pair of constants (integer or real), separated by a comma and enclosed in parentheses. Examples are:

```
(2, -3)
(1., 9.9E-1)
```

The first number denotes the real part and the second the imaginary part.

The fifth type is logical constants. These can only have one of two values:

```
.TRUE.
.FALSE.
```

Note that the dots enclosing the letters are required.

The last type is character constants. These are most often used as an array of characters, called a string. These consist of an arbitrary sequence of characters enclosed in apostrophes (single quotes):

```
'ABC'
'Anything goes!'
'It is a nice day'
```

Strings and character constants are case sensitive. A problem arises if you want to have an apostrophe in the string itself. In this case, you should double the apostrophe:

```
'It''s a nice day'
```

Expressions

The simplest expressions are of the form

```
operand operator operand
```

and an example is

```
x + y
```

The result of an expression is itself an operand, hence we can nest expressions together like

$$x + 2 * y$$

This raises the question of precedence: Does the last expression mean $x + (2*y)$ or $(x+2)*y$? The precedence of arithmetic operators in Fortran 77 are (from highest to lowest):

- ** {exponentiation}
- *,/ {multiplication, division}
- +,- {addition, subtraction}

All these operators are calculated left-to-right, except the exponentiation operator **, which has right-to-left precedence. If you want to change the default evaluation order, you can use parentheses.

The above operators are all binary operators. there is also the unary operator - for negation, which takes precedence over the others. Hence an expression like $-x+y$ means what you would expect.

Extreme caution must be taken when using the division operator, which has a quite different meaning for integers and reals. If the operands are both integers, an integer division is performed, otherwise a real arithmetic division is performed. For example, $3/2$ equals 1, while $3./2.$ equals 1.5.

Assignment

The assignment has the form

$$\text{variable_name} = \text{expression}$$

The interpretation is as follows: Evaluate the right hand side and assign the resulting value to the variable on the left. The expression on the right may contain other variables, but these never change value! For example,

$$\text{area} = \text{pi} * \text{r}^{**2}$$

does not change the value of pi or r, only area.

Type conversion

When different data types occur in the same expression, type conversion has to take place, either explicitly or implicitly. Fortran will do some type conversion implicitly. For example,

$$\begin{aligned} &\text{real } x \\ &x = x + 1 \end{aligned}$$

will convert the integer one to the real number one, and has the desired effect of incrementing x by one. However, in more complicated expressions, it is good programming practice to force the necessary type conversions explicitly. For numbers, the following functions are available:

int
real
dble
ichar
char

The first three have the obvious meaning. ichar takes a character and converts it to an integer, while char does exactly the opposite.

Example: How to multiply two real variables x and y using double precision and store the result in the double precision variable w:

```
w = dble(x)*dble(y)
```

Note that this is different from

```
w = dble(x*y)
```

7. Logical expressions

Logical expressions can only have the value `.TRUE.` or `.FALSE.`. A logical expression can be formed by comparing arithmetic expressions using the following relational operators:

`.LT.` meaning `<` (less than)
`.LE.` `<=` (less than or equal to)
`.GT.` `>` (greater than)
`.GE.` `>=` (greater than or equal to)
`.EQ.` `=` (equal to)
`.NE.` `/=` (not equal to)

So you cannot use symbols like `<` or `=` for comparison in Fortran 77, but you have to use the correct two-letter abbreviation enclosed by dots! (Such symbols are allowed in Fortran 90, though.)

Logical expressions can be combined by the logical operators `.AND.` `.OR.` `.NOT.` which have the obvious meaning.

Logical variables and assignment

Truth values can be stored in logical variables. The assignment is analagous to the arithmetic assignment. Example:


```
logical a, b
a = .TRUE.
b = a .AND. 3 .LT. 5/2
```

The order of precedence is important, as the last example shows. The rule is that arithmetic expressions are evaluated first, then relational operators, and finally logical operators. Hence b will be assigned .FALSE. in the example above.

Logical variables are seldom used in Fortran. But logical expressions are frequently used in conditional statements like the if statement.

8. The if statements

An important part of any programming language are the conditional statements. The most common such statement in Fortran is the ifstatement, which actually has several forms. The simplest one is the logical if statement:

```
if (logical expression) executable statement
```

This has to be written on one line. This example finds the absolute value of x:

```
if (x .LT. 0) x = -x
```

If more than one statement should be executed inside the if, then the following syntax should be used:

```
if (logical expression) then
  statements
endif
```

The most general form of the if statement has the following form:

```
if (logical expression) then
  statements
elseif (logical expression) then
  statements
:
:
else
  statements
endif
```

The execution flow is from top to bottom. The conditional expressions are evaluated in sequence until one is found to be true. Then the associated code is executed and the control jumps to the next statement after the endif.

Nested if statements

if statements can be nested in several levels. To ensure readability, it is important to

use proper indentation. Here is an example:

```
if (x .GT. 0) then
  if (x .GE. y) then
    write(*,*) 'x is positive and x >= y'
  else
    write(*,*) 'x is positive but x < y'
  endif
elseif (x .LT. 0) then
  write(*,*) 'x is negative'
else
  write(*,*) 'x is zero'
endif
```

You should avoid nesting many levels of if statements since things get hard to follow.

9. Loops

For repeated execution of similar things, loops are used. If you are familiar with other programming languages you have probably heard about for-loops, while-loops, and until-loops. Fortran 77 has only one loop construct, called the do-loop. The do-loop corresponds to what is known as a for-loop in other languages. Other loop constructs have to be simulated using the if and goto statements.

do-loops

The do-loop is used for simple counting. Here is a simple example that prints the cumulative sums of the integers from 1 through n (assume n has been assigned a value elsewhere):

```
integer i, n, sum

sum = 0
do 10 i = 1, n
  sum = sum + i
  write(*,*) 'i =', i
  write(*,*) 'sum =', sum
10 continue
```

The number 10 is a statement label. Typically, there will be many loops and other statements in a single program that require a statement label. The programmer is responsible for assigning a unique number to each label in each program (or subprogram). Recall that column positions 2-5 are reserved for statement labels. The numerical value of statement labels have no significance, so any integer numbers can be used. Typically, most programmers increment labels by 10 at a time.

The variable defined in the do-statement is incremented by 1 by default. However, you can define any other integer to be the step. This program segment prints the even numbers between 1 and 10 in decreasing order:

```
integer i

do 20 i = 10, 1, -2
  write(*,*) 'i =', i
20 continue
```

The general form of the do loop is as follows:

```
do label var = expr1, expr2, expr3
  statements
label continue
```

var is the loop variable (often called the loop index) which must be integer. expr1 specifies the initial value of var, expr2 is the terminating bound, and expr3 is the increment (step).

Note: The do-loop variable must never be changed by other statements within the loop! This will cause great confusion.

Many Fortran 77 compilers allow do-loops to be closed by the enddo statement. The advantage of this is that the statement label can then be omitted since it is assumed that an enddo closes the nearest previous do statement. The enddo construct is widely used, but it is not a part of ANSI Fortran 77.

while-loops

The most intuitive way to write a while-loop is

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

or alternatively,

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

The statements in the body will be repeated as long as the condition in the while statement is true. Even though this syntax is accepted by many compilers, it is not ANSI Fortran 77. The correct way is to use if and goto:

```
label if (logical expr) then
  statements
  goto label
endif
```

Here is an example that calculates and prints all the powers of two that are less than or equal to 100:

```
integer n

n = 1
10 if (n .le. 100) then
    n = 2*n
    write (*,*) n
    goto 10
endif
```

until-loops

If the termination criterium is at the end instead of the beginning, it is often called an until-loop. The pseudocode looks like this:

```
do
    statements
until (logical expr)
```

Again, this should be implemented in Fortran 77 by using if and goto:

```
label continue
    statements
    if (logical expr) goto label
```

Note that the logical expression in the latter version should be the negation of the expression given in the pseudocode!

Loops in Fortran 90

Fortran 90 has adopted the do-endo construct as its loop construct. So our "counting down in twos" example will look like this:

```
do i = 10, 1, -2
    write(*,*) 'i =', i
end do
```

For while and until loops you also use the do-endo construct, but you have to add a conditional exit statement. The general case is:

```
do
    statements
    if (logical expr) exit
    statements
end do
```

If you have the exit condition at the beginning it is a while loop, and if it is at the end you have an until loop.

10. Arrays

Many scientific computations use vectors and matrices. The data type Fortran uses for representing such objects is the array. A one-dimensional array corresponds to a vector, while a two-dimensional array corresponds to a matrix. To fully understand how this works in Fortran 77, you will have to know not only the syntax for usage, but also how these objects are stored in memory in Fortran 77.

One-dimensional arrays

The simplest array is the one-dimensional array, which is just a linear sequence of elements stored consecutively in memory. For example, the declaration

```
real a(20)
```

declares `a` as a real array of length 20. That is, `a` consists of 20 real numbers stored contiguously in memory. By convention, Fortran arrays are indexed from 1 and up. Thus the first number in the array is denoted by `a(1)` and the last by `a(20)`. However, you may define an arbitrary index range for your arrays using the following syntax:

```
real b(0:19), weird(-162:237)
```

Here, `b` is exactly similar to `a` from the previous example, except the index runs from 0 through 19. `weird` is an array of length $237 - (-162) + 1 = 400$.

The type of an array element can be any of the basic data types. Examples:

```
integer i(10)
logical aa(0:1)
double precision x(100)
```

Each element of an array can be thought of as a separate variable. You reference the i 'th element of array `a` by `a(i)`. Here is a code segment that stores the 10 first square numbers in the array `sq`:

```
integer i, sq(10)

do 100 i = 1, 10
    sq(i) = i**2
100 continue
```

A common bug in Fortran is that the program tries to access array elements that are out of bounds or undefined. This is the responsibility of the programmer, and the Fortran compiler will not detect any such bugs!

Two-dimensional arrays

Matrices are very important in linear algebra. Matrices are usually represented by two-dimensional arrays. For example, the declaration

```
real A(3,5)
```

defines a two-dimensional array of $3 \times 5 = 15$ real numbers. It is useful to think of the first index as the row index, and the second as the column index. Hence we get the graphical picture:

```
(1,1) (1,2) (1,3) (1,4) (1,5)
(2,1) (2,2) (2,3) (2,4) (2,5)
(3,1) (3,2) (3,3) (3,4) (3,5)
```

Two-dimensional arrays may also have indices in an arbitrary defined range. The general syntax for declarations is:

```
name (low_index1 : hi_index1, low_index2 : hi_index2)
```

The total size of the array is then

```
size = (hi_index1-low_index1+1)*(hi_index2-low_index2+1)
```

It is quite common in Fortran to declare arrays that are larger than the matrix we want to store. (This is because Fortran does not have dynamic storage allocation.) This is perfectly legal. Example:

```
real A(3,5)
integer i,j
c
c We will only use the upper 3 by 3 part of this array.
c
do 20 j = 1, 3
  do 10 i = 1, 3
    a(i,j) = real(i)/real(j)
  10 continue
20 continue
```

The elements in the submatrix $A(1:3,4:5)$ are undefined. Do not assume these elements are initialized to zero by the compiler (some compilers will do this, but not all).

Storage format for 2-dimensional arrays

Fortran stores higher dimensional arrays as a contiguous linear sequence of elements. It is important to know that 2-dimensional arrays are stored by column. So in the above example, array element (1,2) will follow element (3,1). Then follows the rest of the

second column, thereafter the third column, and so on.

Consider again the example where we only use the upper 3 by 3 submatrix of the 3 by 5 array A(3,5). The 9 interesting elements will then be stored in the first nine memory locations, while the last six are not used. This works out neatly because the leading dimension is the same for both the array and the matrix we store in the array. However, frequently the leading dimension of the array will be larger than the first dimension of the matrix. Then the matrix will not be stored contiguously in memory, even if the array is contiguous. For example, suppose the declaration was A(5,3) instead. Then there would be two "unused" memory cells between the end of one column and the beginning of the next column (again we are assuming the matrix is 3 by 3).

This may seem complicated, but actually it is quite simple when you get used to it. If you are in doubt, it can be useful to look at how the address of an array element is computed. Each array will have some memory address assigned to the beginning of the array, that is element (1,1). The address of element (i,j) is then given by

$$\text{addr}[A(i,j)] = \text{addr}[A(1,1)] + (j-1)*\text{lda} + (i-1)$$

where lda is the leading (i.e. column) dimension of A. Note that lda is in general different from the actual matrix dimension. Many Fortran errors are caused by this, so it is very important you understand the distinction!

Multi-dimensional arrays

Fortran 77 allows arrays of up to seven dimensions. The syntax and storage format are analogous to the two-dimensional case, so we will not spend time on this.

The dimension statement

There is an alternate way to declare arrays in Fortran 77. The statements

```
real A, x
dimension x(50)
dimension A(10,20)
```

are equivalent to

```
real A(10,20), x(50)
```

This dimension statement is considered old-fashioned style today.

11. Subprograms

When a program is more than a few hundred lines long, it gets hard to follow. Fortran codes that solve real engineering problems often have tens of thousands of lines. The only way to handle such big codes, is to use a modular approach and split the program into many separate smaller units called subprograms.

A subprogram is a (small) piece of code that solves a well defined subproblem. In a large program, one often has to solve the same subproblems with many different data. Instead of replicating code, these tasks should be solved by subprograms. The same subprogram can be invoked many times with different input data.

Fortran has two different types of subprograms, called functions and subroutines.

Functions

Fortran functions are quite similar to mathematical functions: They both take a set of input arguments (parameters) and return a value of some type. In the preceding discussion we talked about user defined subprograms. Fortran 77 also has some built-in functions.

A simple example illustrates how to use a function:

```
x = cos(pi/3.0)
```

Here cos is the cosine function, so x will be assigned the value 0.5 (if pi has been correctly defined; Fortran 77 has no built-in constants). There are many built-in functions in Fortran 77. Some of the most common are:

abs	absolute value
min	minimum value
max	maximum value
sqrt	square root
sin	sine
cos	cosine
tan	tangent
atan	arctangent
exp	exponential (natural)
log	logarithm (natural)

In general, a function always has a type. Most of the built-in functions mentioned above, however, are generic. So in the example above, pi and x could be either of type real or double precision. The compiler would check the types and use the correct version of cos (real or double precision). Unfortunately, Fortran is not really a polymorphic language so in general you have to be careful to match the types of your variables and your functions!

Now we turn to the user-written functions. Consider the following problem: A meteorologist has studied the precipitation levels in the Bay Area and has come up with a model $r(m,t)$ where r is the amount of rain, m is the month, and t is a scalar parameter that depends on the location. Given the formula for r and the value of t , compute the annual rainfall.

The obvious way to solve the problem is to write a loop that runs over all the months

and sums up the values of r . Since computing the value of r is an independent subproblem, it is convenient to implement it as a function. The following main program can be used:

```
program rain
real r, t, sum
integer m

read (*,*) t
sum = 0.0
do 10 m = 1, 12
    sum = sum + r(m, t)
10 continue
write (*,*) 'Annual rainfall is ', sum, 'inches'

stop
end
```

In addition, the function r has to be defined as a Fortran function. The formula the meteorologist came up with was

$$r(m,t) = t/10 * (m^{**2} + 14*m + 46) \text{ if this is positive}$$
$$r(m,t) = 0 \text{ otherwise}$$

The corresponding Fortran function is

```
real function r(m,t)
integer m
real t

r = 0.1*t * (m**2 + 14*m + 46)
if (r .LT. 0) r = 0.0

return
end
```

We see that the structure of a function closely resembles that of the main program. The main differences are:

- * Functions have a type. This type must also be declared in the calling program.
- * The return value should be stored in a variable with the same name as the function.
- * Functions are terminated by the return statement instead of stop.

To sum up, the general syntax of a Fortran 77 function is:

```
type function name (list-of-variables)
```

```
declarations
statements
return
end
```

The function has to be declared with the correct type in the calling program unit. The function is then called by simply using the function name and listing the parameters in parenthesis.

Subroutines

A Fortran function can essentially only return one value. Often we want to return two or more values (or sometimes none!). For this purpose we use the subroutine construct. The syntax is as follows:

```
subroutine name (list-of-arguments)
declarations
statements
return
end
```

Note that subroutines have no type and consequently should not (cannot) be declared in the calling program unit.

We give an example of a very simple subroutine. The purpose of the subroutine is to swap two integers.

```
subroutine iswap (a, b)
integer a, b
c Local variables
integer tmp

tmp = a
a = b
b = tmp

return
end
```

Note that there are two blocks of variable declarations here. First, we declare the input/output parameters, i.e. the variables that are common to both the caller and the callee. Afterwards, we declare the local variables, i.e. the variables that can only be used within this subprogram. We can use the same variable names in different subprograms and the compiler will know that they are different variables that just happen to have the same names.

Call-by-reference

Fortran 77 uses the so-called call-by-reference paradigm. This means that instead of

just passing the values of the function/subroutine arguments (call-by-value), the memory address of the arguments (pointers) are passed instead. A small example should show the difference:

```
program callex
integer m, n
c
m = 1
n = 2

call iswap(m, n)
write(*,*) m, n

stop
end
```

The output from this program is "2 1", just as one would expect. However, if Fortran 77 had been using call-by-value then the output would have been "1 2", i.e. the variables m and n were unchanged! The reason for this is that only the values of m and n had been copied to the subroutine iswap, and even if a and b were swapped inside the subroutine the new values would not have been passed back to the main program.

In the above example, call-by-reference was exactly what we wanted. But you have to be careful about this when writing Fortran code, because it is easy to introduce undesired side effects. For example, sometimes it is tempting to use an input parameter in a subprogram as a local variable and change its value. You should never do this since the new value will then propagate back to the calling program with an unexpected value!

We will come back to this issue in a later section on passing arrays as arguments (parameters).

12. Simple I/O

An important part of any computer program is to handle input and output. In our examples so far, we have already used the two most common Fortran constructs for this: read and write. Fortran I/O can be quite complicated, so we will only describe some simpler cases in this tutorial.

Read and write

Read is used for input, while write is used for output. A simple form is

```
read (unit no, format no) list-of-variables
write(unit no, format no) list-of-variables
```

The unit number can refer to either standard input, standard output, or a file. This will be described in later section. The format number refers to a label for a format statement,

which will be described shortly.

It is possible to simplify these statements further by using asterisks (*) for some arguments, like we have done in all our examples so far. This is sometimes called list directed read/write.

```
read (*, *) list-of-variables  
write(*, *) list-of-variables
```

The first statement will read values from the standard input and assign the values to the variables in the variable list, while the second one writes to the standard output.

Examples

Here is a code segment from a Fortran program:

```
integer m, n  
real x, y  
  
read(*, *) m, n  
read(*, *) x, y
```

We give the input through standard input (possibly through a data file directed to standard input). A data file consists of records according to traditional Fortran terminology. In our example, each record contains a number (either integer or real). Records are separated by either blanks or commas. Hence a legal input to the program above would be:

```
-1 100  
-1.0 1e+2
```

Or, we could add commas as separators:

```
-1, 100  
-1.0, 1e+2
```

Note that Fortran 77 input is line sensitive, so it is important to have the right number of input elements (records) on each line. For example, if we gave the input all on one line as

```
-1, 100, -1.0, 1e+2
```

then m and n would be assigned the values -1 and 100 respectively, but the last two values would be discarded, leaving x and y undefined.

Other versions

For simple list-directed I/O it is possible to use the alternate syntax

```
read *, list-of-variables  
print *, list-of-variables
```

which has the same meaning as the list-directed read and write statements described earlier. This version always reads/writes to standard input/output so the * corresponds to the format.

13. Format statements

So far we have only showed free format input/output. This uses a set of default rules for how to output values of different types (integers, reals, characters, etc.). Often the programmer wants to specify some particular input or output format, e.g. how many decimals in real numbers. For this purpose Fortran 77 has the format statement. The same format statements are used for both input and output.

Syntax

```
write(*, label) list-of-variables  
label format format-code
```

A simple example demonstrates how this works. Say you have an integer variable you want to print in a field 4 characters wide and a real number you want to print in fixed point notation with 3 decimal places.

```
write(*, 900) i, x  
900 format (I4,F8.3)
```

The format label 900 is chosen somewhat arbitrarily, but it is common practice to number format statements with higher numbers than the control flow labels. After the keyword format follows the format codes enclosed in parenthesis. The code I4 stands for an integer with width four, while F8.3 means that the number should be printed using fixed point notation with field width 8 and 3 decimal places.

The format statement may be located anywhere within the program unit. There are two programming styles: Either the format statement follows directly after the read/write statement, or all the format statements are grouped together at the end of the (sub-)program.

Common format codes

The most common format code letters are:

- A - text string
- D - double precision numbers, exponent notation
- E - real numbers, exponent notation
- F - real numbers, fixed point format
- I - integer
- X - horizontal skip (space)
- / - vertical skip (newline)

The format code F (and similarly D, E) has the general form Fw.d where w is an integer constant denoting the field width and d is an integer constant denoting the number of significant digits.

For integers only the field width is specified, so the syntax is Iw. Similarly, character strings can be specified as Aw but the field width is often dropped.

If a number or string does not fill up the entire field width, spaces will be added. Usually the text will be adjusted to the right, but the exact rules vary among the different format codes.

For horizontal spacing, the nX code is often used. This means n horizontal spaces. If n is omitted, n=1 is assumed. For vertical spacing (newlines), use the code /. Each slash corresponds to one newline. Note that each read or write statement by default ends with a newline (here Fortran differs from C).

Some examples

This piece of Fortran code

```
x = 0.025
write(*,100) 'x=', x
100 format (A,F)
write(*,110) 'x=', x
110 format (A,F5.3)
write(*,120) 'x=', x
120 format (A,E)
write(*,130) 'x=', x
130 format (A,E8.1)
```

produces the following output when we run it:

```
x=   0.0250000
x=0.025
x=  0.2500000E-01
x=  0.3E-01
```

Note how blanks are automatically padded on the left and that the default field width for real numbers is usually 14. We see that Fortran 77 follows the rounding rule that digits 0-4 are rounded downwards while 5-9 is rounded upwards.

In this example each write statement used a different format statement. But it is perfectly fine to use the same format statement from many different write statements. In fact, this is one of the main advantages of using format statements. This feature is handy when you print tables for instance, and want each row to have the same format.

Format strings in read/write statements

Instead of specifying the format code in a separate format statement, one can give the format code in the read/write statement directly. For example, the statement

```
write (*,'(A, F8.3)') 'The answer is x = ', x
```

is equivalent to

```
write (*,990) 'The answer is x = ', x
990 format (A, F8.3)
```

Sometimes text strings are given in the format statements, e.g. the following version is also equivalent:

```
write (*,999) x
999 format ('The answer is x = ', F8.3)
```

Implicit loops and repeat counts

Now let us do a more complicated example. Say you have a two-dimensional array of integers and want to print the upper left 5 by 10 submatrix with 10 values each on 5 rows. Here is how:

```
do 10 i = 1, 5
  write(*,1000) (a(i,j), j=1,10)
10 continue
1000 format (I6)
```

We have an explicit do loop over the rows and an implicit loop over the column index j.

Often a format statement involves repetition, for example

```
950 format (2X, I3, 2X, I3, 2X, I3, 2X, I3)
```

There is a shorthand notation for this:

```
950 format (4(2X, I3))
```

It is also possible to allow repetition without explicitly stating how many times the format should be repeated. Suppose you have a vector where you want to print the first 50 elements, with ten elements on each line. Here is one way:

```
write(*,1010) (x(i), i=1,50)
1010 format (10I6)
```

The format statements says ten numbers should be printed. But in the write statement we try to print 50 numbers. So after the ten first numbers have been printed, the same format statement is automatically used for the next ten numbers and so on.

14. File I/O

So far we have assumed that the input/output has been to the standard input or the standard output. It is also possible to read or write from files which are stored on some external storage device, typically a disk (hard disk, floppy) or a tape. In Fortran each file is associated with a unit number, an integer between 1 and 99. Some unit numbers are reserved: 5 is standard input, 6 is standard output.

Opening and closing a file

Before you can use a file you have to open it. The command is

```
open (list-of-specifiers)
```

where the most common specifiers are:

```
[UNIT=] u  
IOSTAT= ios  
ERR= err  
FILE= fname  
STATUS= sta  
ACCESS= acc  
FORM= frm  
RECL= rl
```

The unit number *u* is a number in the range 9-99 that denotes this file (the programmer may chose any number but he/she has to make sure it is unique).

ios is the I/O status identifier and should be an integer variable. Upon return, *ios* is zero if the stement was successful and returns a non-zero value otherwise.

err is a label which the program will jump to if there is an error.

fname is a character string denoting the file name.

sta is a character string that has to be either NEW, OLD or SCRATCH. It shows the prior status of the file. A scrath file is a file that is created and deleted when the file is closed (or the program ends).

acc must be either SEQUENTIAL or DIRECT. The default is SEQUENTIAL.

frm must be either FORMATTED or UNFORMATTED. The default is UNFORMATTED.

rl specifies the length of each record in a direct-access file.

For more details on these specifiers, see a good Fortran 77 book.

After a file has been opened, you can access it by read and write statements. When you are done with the file, it should be closed by the statement

```
close ([UNIT=]u[,IOSTAT=ios,ERR=err,STATUS=sta])
```

where, as usual, the parameters in brackets are optional.

Read and write revisited

The only necessary change from our previous simplified read/write statements, is that the unit number must be specified. But frequently one wants to add more specifiers.

Here is how:

```
read ([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s)
write([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s)
```

where most of the specifiers have been described above. The END=s specifier defines which statement label the program jumps to if it reaches end-of-file.

Example

You are given a data file with xyz coordinates for a bunch of points. The number of points is given on the first line. The file name of the data file is points.dat. The format for each coordinate is known to be F10.4. Here is a short program that reads the data into 3 arrays x,y,z:

```
program inpdatt
c
c This program reads n points from a data file and stores them in
c 3 arrays x, y, z.
c
integer nmax, u
parameter (nmax=1000, u=20)
real x(nmax), y(nmax), z(nmax)

c Open the data file
open (u, FILE='points.dat', STATUS='OLD')

c Read the number of points
read(u,*) n
if (n.GT.nmax) then
write(*,*) 'Error: n = ', n, 'is larger than nmax =', nmax
goto 9999
endif

c Loop over the data points
```

```

do 10 i= 1, n
  read(u,100) x(i), y(i), z(i)
10 enddo
100 format (3(F10.4))

```

```

c Close the file
  close (u)

```

```

c Now we should process the data somehow...
c (missing part)

```

```

9999 stop
  end

```

15. Arrays in subprograms

Fortran subprogram calls are based on call by reference. This means that the calling parameters are not copied to the called subprogram, but rather that the addresses of the parameters (variables) are passed. This saves a lot of memory space when dealing with arrays. No extra storage is needed as the subroutine operates on the same memory locations as the calling (sub-)program. However, you as a programmer has to know about this and take it into account.

It is possible to declare local arrays in Fortran subprograms, but this feature is rarely used. Typically, all arrays are declared (and dimensioned) in the main program and then passed on to the subprograms as needed.

Variable length arrays

A basic vector operation is the saxpy operation. This calculates the expression

$$y := \alpha x + y$$

where α is a scalar but x and y are vectors. Here is a simple subroutine for this:

```

subroutine saxpy (n, alpha, x, y)
  integer n
  real alpha, x(*), y(*)
c
c Saxpy: Compute y := alpha*x + y,
c where x and y are vectors of length n (at least).
c
c Local variables
  integer i
c
  do 10 i = 1, n
    y(i) = alpha*x(i) + y(i)
10 continue

```

```
c
  return
end
```

The only new feature here is the use of the asterisk in the declarations $x(*)$ and $y(*)$. This notation says that x and y are arrays of arbitrary length. The advantage of this is that we can use the same subroutine for all vector lengths. Recall that since Fortran is based on call-by-reference, no additional space is allocated but rather the subroutine works directly on the array elements from the calling routine/program. It is the responsibility of the programmer to make sure that the vectors x and y really have been declared to have length n or more in some other program unit. A common error in Fortran 77 occurs when you try to access out-of-bounds array elements.

We could also have declared the arrays like this:

```
real x(n), y(n)
```

Most programmers prefer to use the asterisk notation to emphasize that the "real array length" is unknown. Some old Fortran 77 programs may declare variable length arrays like this:

```
real x(1), y(1)
```

This is legal syntax even if the array lengths are greater than one! But this is poor programming style and is strongly discouraged.

Passing subsections of arrays

Next we want to write a subroutine for matrix-vector multiplication. There are two basic ways to do this, either by using inner products or saxpy operations. Let us be modular and re-use the saxpy code from the previous section. A simple code is given below.

```
subroutine matvec (m, n, A, lda, x, y)
  integer m, n, lda
  real x(*), y(*), A(lda,*)
c
c Compute  $y = A*x$ , where  $A$  is  $m$  by  $n$  and stored in an array
c with leading dimension  $lda$ .
c
c Local variables
  integer i, j
c Initialize y
  do 10 i = 1, m
    y(i) = 0.0
  10 continue
```

```

c Matrix-vector product by saxpy on columns in A.
c Notice that the length of each column of A is m, not n!
  do 20 j = 1, n
    call saxpy (m, x(j), A(1,j), y)
  20 continue

  return
end

```

There are several important things to note here. First, note that even if we have written the code as general as possible to allow for arbitrary dimensions m and n , we still need to specify the leading dimension of the matrix A . The variable length declaration (*) can only be used for the last dimension of an array! The reason for this is the way Fortran 77 stores multidimensional arrays (see the section on arrays).

When we compute $y = A*x$ by saxpy operations, we need to access columns of A . The j 'th column of A is $A(1:m,j)$. However, in Fortran 77 we cannot use such subarray syntax (but it is encouraged in Fortran 90!). So instead we provide a pointer to the first element in the column, which is $A(1,j)$ (it is not really a pointer, but it may be helpful to think of it as if it were). We know that the next memory locations will contain the succeeding array elements in this column. The saxpy subroutine will treat $A(1,j)$ as the first element of a vector, and does not know that this vector happens to be a column of a matrix.

Finally, note that we have stuck to the convention that matrices have m rows and n columns. The index i is used as a row index (1 to m) while the index j is used as a column index (1 to n). Most Fortran programs handling linear algebra use this notation and it makes it a lot easier to read the code!

Different dimensions

Sometimes it can be beneficial to treat a 1-dimensional array as a 2-dimensional array and vice versa. This is fairly simple to do in Fortran 77, some will say it is too easy!

Let us look at a very simple example. Another basic vector operation is scaling, i.e. multiplying each element in a vector by the same constant. Here is a subroutine for this:

```

subroutine scale(n, alpha, x)
  integer n
  real alpha, x(*)
c
c Local variables
  integer i

  do 10 i = 1, n
    x(i) = alpha * x(i)
  10 continue

```

```
return
end
```

Now suppose we have a m by n matrix we want to scale. Instead of writing a new subroutine for this, we can simply treat the matrix as a vector and use the subroutine `scale`. A simple version is given first:

```
integer m, n
parameter (m=10, n=20)
real alpha, A(m,n)
```

c Some statements here define A...

```
c Now we want to scale A
  call scale(m*n, alpha, A)
```

Note that this example works because we assume the declared dimension of A equals the actual dimension of the matrix stored in A . This does not hold in general. Often the leading dimension lda is different from the actual dimension m , and great care must be taken to handle this correctly. Here is a more robust subroutine for scaling a matrix that uses the subroutine `scale`:

```
subroutine mscale(m, n, alpha, A, lda)
integer m, n, lda
real alpha, A(lda,*)
c
c Local variables
integer j

do 10 j = 1, n
  call scale(m, alpha, A(1,j) )
10 continue

return
end
```

This version works even when m is not equal to lda since we scale one column at a time and only process the m first elements of each column (leaving the rest untouched).

16. Common blocks

Fortran 77 has no global variables, i.e. variables that are shared among several program units (subroutines). The only way to pass information between subroutines we have seen so far, is to use the subroutine parameter list. Sometimes this is inconvenient, e.g. when many subroutines share a large set of parameters. In such cases one can use a common block. This is a way to specify that certain variables should be shared among certain subroutines. But in general, the use of common blocks

should be minimized.

Example

Suppose you have two parameters alpha and beta that many of your subroutines need. The following example shows how it can be done using common blocks.

```
program main
some declarations
real alpha, beta
common /coeff/ alpha, beta

statements
stop
end

subroutine sub1 (some arguments)
declarations of arguments
real alpha, beta
common /coeff/ alpha, beta

statements
return
end

subroutine sub2 (some arguments)
declarations of arguments
real alpha, beta
common /coeff/ alpha, beta

statements
return
end
```

Here we define a common block with the name coeff. The content of the common block is the two variables alpha and beta. A common block can contain as many variables as you like. They do not need to all have the same type. Every subroutine that wants to use any of the variables in the common block has to declare the whole block.

Note that in this example we could easily have avoided common blocks by passing alpha and beta as parameters (arguments). A good rule is to try to avoid common blocks if possible. However, there are a few rare cases where there is no other solution.

Syntax

```
common / name / list-of-variables
```

You should know that

- * The common statement should appear together with the variable declarations, before the executable statements.

- * Different common blocks must have different names (just like variables). A variable can belong to more than one common block.

- * The variables in a common block do not need to have the same names each place they occur (although it is a good idea to do so), but they must be listed in the same order and have the same type.

To illustrate this, look at the following continuation of our example:

```
subroutine sub3 (some arguments)
  declarations of arguments
  real a, b
  common /coeff/ a, b

  statements
  return
end
```

This declaration is equivalent to the previous version that used alpha and beta. It is recommended that you always use the same variable names for the same common block to avoid confusion. Here is a dreadful example:

```
subroutine sub4 (some arguments)
  declarations of arguments
  real alpha, beta
  common /coeff/ beta, alpha

  statements
  return
end
```

Now alpha is the beta from the main program and vice versa. If you see something like this, it is probably a mistake. Such bugs are very hard to find.

Arrays in common blocks

Common blocks can include arrays, too. But again, this is not recommended. The major reason is flexibility. An example shows why this is such a bad idea. Suppose we have the following declarations in the main program:

```
program main
  integer nmax
  parameter (nmax=20)
  integer n
```

```
real A(nmax, nmax)
common /matrix/ A, n, nmax
```

This common block contains first all the elements of A, then the integers n and nmax. Now assume you want to use the matrix A in some subroutines. Then you have to include the same declarations in all these subroutines, e.g.

```
subroutine sub1 (...)
integer nmax
parameter (nmax=20)
integer n
real A(nmax, nmax)
common /matrix/ A, n, nmax
```

Arrays with variable dimensions cannot appear in common blocks, thus the value of nmax has to be exactly the same as in the main program. Recall that the size of a matrix has to be known at compile time, hence nmax has to be defined in a parameter statement. It would be tempting to delete the parameter statement in the subroutine since nmax belongs to the common block, but this would be illegal.

This example shows there is usually nothing to gain by putting arrays in common blocks. Hence the preferred method in Fortran 77 is to pass arrays as arguments to subroutines (along with the leading dimensions).

17. data and block data

The data statement

The data statement is another way to input data that are known at the time when the program is written. It is similar to the assignment statement. The syntax is:

```
data list-of-variables/ list-of-values/, ...
```

where the three dots means that this pattern can be repeated. Here is an example:

```
data m/10/, n/20/, x/2.5/, y/2.5/
```

We could also have written this

```
data m,n/10,20/, x,y/2*2.5/
```

We could have accomplished the same thing by the assignments

```
m = 10
n = 20
x = 2.5
y = 2.5
```


The data statement is more compact and therefore often more convenient. Notice especially the shorthand notation for assigning identical values repeatedly.

The data statement is performed only once, right before the execution of the program starts. For this reason, the data statement is mainly used in the main program and not in subroutines.

The data statement can also be used to initialize arrays (vectors, matrices). This example shows how to make sure a matrix is all zeros when the program starts:

```
real A(10,20)
data A/ 200 * 0.0/
```

Some compilers will automatically initialize arrays like this but not all, so if you rely on array elements to be zero it is a good idea to follow this example. Of course you can initialize arrays to other values than zero. You may even initialize individual elements:

```
data A(1,1)/ 12.5/, A(2,1)/ -33.3/, A(2,2)/ 1.0/
```

Or you can list all the elements for small arrays like this:

```
integer v(5)
real B(2,2)
data v/10,20,30,40,50/, B/1.0,-3.7,4.3,0.0/
```

The values for two-dimensional arrays will be assigned in column-first order as usual.

The block data statement

The data statement cannot be used for variables contained in a common block. There is a special "subroutine" for this purpose, called block data. It is not really a subroutine, but it looks a bit similar because it is given as a separate program unit. Here is an example:

```
block data
integer nmax
parameter (nmax=20)
real v(nmax), alpha, beta
common /vector/v,alpha,beta
data v/20*100.0/, alpha/3.14/, beta/2.71/
end
```

Just as the data statement, block data is executed once before the execution of the main program starts. The position of the block data "subroutine" in the source code is irrelevant (as long as it is not nested inside the main program or a subprogram).

18. Fortran programming style

There are many different programming styles, but we will try to give some general

guidelines that are fairly non-controversial.

Portability

To ensure portability, use only standard Fortran 77. The only exception we have allowed in this tutorial is to use lower case letters today.

Program structure

The overall program structure should be modular. Each subprogram should solve a well-defined task. Many people prefer to write each subprogram in a separate file.

Comments

Let us repeat this: Write legible code, but also add comments in the source explaining what is happening! It is especially important to have a good header for each subprogram that explains each input/output argument and what the subprogram does.

Indentation

Always use proper indentation for loops and if blocks as demonstrated in this tutorial.

Variables

Always declare all variables. Implicit type declaration is bad! Try to stick to maximum 6 characters for variable names, or at the very least make sure the 6 first characters are unique.

Subprograms

Never let functions have "side effects", i.e. do not change the value of the input parameters. Use subroutines in such cases.

In the declarations, separate parameters, common blocks, and local variables.

Minimize the use of common blocks.

Goto

Minimize the use of goto. Unfortunately it is necessary to use goto in some loops since while is not standard Fortran.

Arrays

In many cases it is best to declare all large arrays in the main program and then pass them as arguments to the various subroutines. This way all the space allocation is done in one place. Remember to pass the leading dimensions. Avoid unnecessary "reshaping" of matrices.

Efficiency concerns

When you have a double loop accessing a two-dimensional array, it is usually best to have the first (row) index in the innermost loop. This is because of the storage scheme in Fortran.

When you have if-then-elseif statements with multiple conditions, try to place the most likely conditions first.

19. Debugging hints

It has been estimated that about 90% of the time it takes to develop commercial software is spent debugging and testing. This shows how important it is to write good code in the first place.

Still, we all discover from time to time. Here are some hints for how to track them down.

Useful compiler options

Most Fortran compilers will have a set of options you can turn on if you like. The following compiler options are specific to the BC-FORTRAN77 compiler, but most compilers will have similar options (although the letters may be different).

You can use /Option instead of -Option.

- D Debug: equivalent to -BVL
- B deBugcode: implies -H and also produces debugger calls for examination of field borders and substrings.
- H parcHeck: produces code for examination of parameter lists and checks of stack overruns
- V Varlist: List of variables for debugger
- L Lineslist: Line number list for debugger
- U Uppercase: Changes lower-case into upper case characters (even in character constants and formats). Without this option xy, XY, Xy and xY are four different variables!
- F Free input: Source program is not in FORTRAN format. Non-standard!
- P Protocol: registry of the compiled lines.
- S Short protocol: One line per module with the size of the stack frame and code in bytes.
- O One: produces one output file with name of the first input file.

Some common errors

Here are some common errors to watch out for:

- * Make sure your lines end at column 72. The rest will be ignored!
- * Do your parameter lists in the calling and the called program match?
- * Do your common blocks match?
- * Have you done integer division when you wanted real division?

Debugging tools

If you have a bug, you have to try to locate it. Syntax errors are easy to find. The problem is when you have run-time errors. The old-fashioned way to find errors is to add write statements in your code and try to track the values of your variables. This is a bit tedious since you have to recompile your source code every time you change the program slightly. BC-FORTRAN77 has a debugger which is a convenient tool. You can step through a program line by line, you can display the values of the variables you want to monitor, and much more. When the program has been compiled with the -D option, a run time error will give an error message and wait for a command. The debugger can also be activated by a key-stroke interrupting the program's execution if the program was created with the B option. The following keys are recognized by the debugger as commands:

T

(Traceback) Look back through the subroutine call stack and find the next higher module with line number if the program was compiled with the -L option.

U

(Untrace) Cancel all trace back actions. This happens automatically when the program is restarted.

V

Display Variables of the current subroutine with their current contents if the program was compiled with the -V option. Move into the last displayed program module when T is pressed.

Z

Terminate program.

(space)

Resume program execution, unless there was a run-time error encountered.

S

Step program until next label, subprogram ELSE, ENDIF or DO.

C

Step subroutine until next CALL or RETURN

Note on Error Messages

Since the BC-FORTRAN77 compiler outputs error messages in German, here are some of the more common error messages:

#	Description
---	-----
200	Unrecognized statement
201	Declaration not allowed here
203	Multiply defined label
252	Illegal expression
253	Equals sign expected
282	Undefined label

The format of the compiler error messages is the error number followed by the error description (in German). On the next line is the string which was being scanned when the error was encountered, followed by the line number where the error was noticed.

A special case are the Input/Output (I/O) errors. They take the form:

IO-Error: Number

Input/output errors may be intercepted by placing the IOSTAT= parameter in the IO statement. The number has the same value as is returned in the IOSTAT variable.

Number:

- 1 End of File
- 2 End of File in middle of record.

- 1 File number already in use (OPEN)
- 2 Too many files (OPEN). Max. is 32
- 3 File name too long (OPEN). Max is 32 characters.
- 5 File number not opened. Only file numbers 5 and 6 can be used with being OPENed
- 6 formatted/unformatted contradiction
- 7 REC parameter for file with ACCESS='SEQUENTIAL'
- 8 BACKSPACE, REWIND, ENDFILE are not possible for files AUX: and PRN:
- 10 Internal I/O is formatted only
- 20 INTEGER overflow
- 21 Data error by the listing generator input, e.g. missing apostrophe for a symbol link
- 22 unexpected end of data
- 23 record too long
- 24 UNFORMATTED SEQUENTIAL read error
- 30 IO list element doesn't match format element.
INTEGER needs I format
REAL, DOUBLE PRECISION (COMPLEX) need one or two F, G, E or D format elements
- 31 Format element L: T or F expected (input)
- 32 format element I: unexpected characters (input). Null characters, signs and numerics allowed.
- 33 format element E, F or D unexpected characters (input null characters, signs, numerics, decimal point, E, e, D or d (for the exponent) allowed.
- 34 too many (in format
- 35 too many) in format
- 36 numeric character in a disallowed place in format
- 37 illegal format element
- 38 decimal point error in F, E, G, or D format element
- 39 4Htext or 'text' in format for READ. In FORTRAN66 this was allowed as a way to read in text but in FORTRAN77 this is no longer allowed.
- 40 exponent too large for the given format.
- 42 format element too long. Must be less than the internal record buffer size (80 characters).
- 43 Internal I/O, record too long.
- 44 TL, record too long.
- 45 zero field length in I, L, F, E, G, D format

1000+n MS-DOS Error n

- 1002 File not found
- 1003 Path not found
- 1004 Too many open files
- 1005 Access denied
- 1008 Not enough free RAM

1012 Invalid access code
1015 Invalid disk drive

Have Fun.

20. Fortran 90 highlights

Free format on source code.

In Fortran 90, you can use either the Fortran 77 input format or free format. If you use free format, the file extension .90 should be used for the file name.

Dynamic allocation and pointers.

It is now possible to allocate storage dynamically. This enables us to finally get rid of all the "work" arrays!

User defined data types.

You can now define your own composite data types, similar to struct in C or record in Pascal.

Modules.

Modules enables you to program in an object oriented style, similar to C++. Modules can also be used to hide global variables, thereby making the Fortran 77 common construct outdated.

Recursive functions.

Now a part of the language.

Built-in array operations.

Statements like $A=0$ and $C=A+B$ are now valid when A and B are arrays. There is also a built-in function for matrix multiplication (matmul).

Operator overloading.

You can define your own meaning of operators like + and = for your own data types (objects).

And there are many more features, too numerous to mention here. Fortran 90 is very different from earlier versions of Fortran. But for backwards compability, Fortran 77 has been included as a subset of Fortran 90.