Fortran 90/95 Programming Manual

fifth revision (2005)

Tanja van Mourik
Chemistry Department
University College London

Fortran 90/95 Programming Manual

Brief History of Fortran

The first FORTRAN (which stands for Formula Translation) compiler was developed in 1957 at IBM. In 1966 the American Standards Association (later the America National Standards Institute, ANSI) released the first standard version of FORTRAN, which later became known as FORTRAN 66. The standard aimed at providing a standardised, portable language, which could easily be transferred from one computer system to the other. In 1977, a revised version of FORTRAN, FORTRAN 77, was completed (the standard was published in 1978). The next version is Fortran 90, which is a major advance over FORTRAN 77. It contains many new features; however, it also contains all of FORTRAN 77. Thus, a standard-conforming FORTRAN 77 program is also a standard-conforming Fortran 90 program. Some of FORTRAN 77's features were identified as "obsolescent", but they were not deleted. Obsolescent features are candidates for deletion in the next standard, and should thus be avoided. The latest standard is Fortran 95. Fortran 95 contains only minor changes to Fortran 90; however, a few of the obsolescent features identified in Fortran 90 were deleted.

Because of the requirement that all of FORTRAN 77's features must be contained in Fortran 90, there are often several ways to do the same thing, which may lead to confusion. For example, an integer-type variable can be declared in FORTRAN 77 format:

```
integer i
```

or in Fortran 90 format:

```
integer :: i
```

In addition, as a legacy from FORTRAN 77, Fortran 90 contains features that are considered bad or unsafe programming practice, but they are not deleted in order to keep the language "backward compatible".

To remedy these problems, a small group of programmers developed the F language. This is basically a subset of Fortran 90, designed to be highly regular and reliable to use. It is much more compact than Fortran 90 (because it does not contain all of FORTRAN 77's features).

There will be a new standard soon, which will be called Fortran 2003 (even though the final international standard will not come out before late 2004). There will be new features in Fortran 2003 (such as support for exception handling, object-oriented programming, and improved interoperability with the C language), but the difference between Fortran 90/95 and Fortran 2000 will not be as large as that between FORTRAN 77 and Fortran 90.

Introduction to the course

This course intends to teach Fortran 90/95, but from the point of view of the F language. Thus, many of the old FORTRAN 77 features will not be discussed, and should not be used in the programs.

It is assumed that you have access to a computer with a Fortran 90 or Fortran 95 compiler. It is strongly recommended to switch on the compiler flag that warns when the compiler encounters source code that does not conform to the Fortran 90 standard, and the flag that shows warning messages. For example:

Silicon Graphics:	f90 –ansi –w2 –o executable-name sourcefile.f90
	Or even better: f90 –ansi –fullwarn –o executable-name sourcefile.f90
Sun:	f90 –ansi

You can check these flags by typing "man f90" or "man f95" (on a Unix system). You may ask someone in your group for help how to use the compiler and editor on the computer you use.

If you have access to emacs or xemacs, and know how to use it (or are willing to invest a bit of time in learning how to use it), it is recommended to use this editor. It will pay off (emacs can format the source code for you, and thus detect program mistakes early on).

Bibliography

Fortran 95 Handbook, complete ISO/ANSI Reference
J.C. Adams, W.S. Brainerd, B.T. Smith, J.L. Wagener, The MIT Press, 1997

The F programming language
M. Metcalf and J. Reid, Oxford University Press, 1996

Programming in Fortran 90
I.M. Smith, John Wiley and Sons, 1995

Migrating to Fortran 90
J.F. Kerrigan, O'Reilly & Associates, Inc., 1993

CONTENTS

1. Getting started

Type, compile and run the following program (call the file hello.f90):

```
program hello
! this programs prints "hello world!" to the screen
   implicit none
   print*, "Hello world!"
end program hello
```

Note the following:

- a program starts with program program_name
- it ends with end program program_name
- print* displays data (in this case, the character string “Hello, world!”) on the screen.
- all characters after the exclamation mark (!) (except in a character string) are ignored by the compiler. It is good programming practice to include comments. Comments can also start after a statement, for example:

  print*, “Hello world!” ! this line prints the message “Hello world!”
- Note the indentation. Indentation is essential to keep a program readable. Additionally, empty lines are allowed and can help to make the program readable.
- Fortran 90 allows both upper and lowercase letters (unlike FORTRAN 77, in which only uppercase was allowed).

2. Types, Variables, Constants, Operators

Names in Fortran 90

A name or “identifier” in Fortran must adhere to fixed rules. They cannot be longer than 31 characters, must be composed of alphanumeric characters (all the letters of the alphabet, and the digits 0 to 9) and underscores (_), and the first character must be a letter. Identifiers are case-insensitive (except in character strings like “Hello world!” in the example above); Thus, PRINT and print are completely identical.

Types

A *variable* is a data object whose value can be defined and redefined (in contrast to *constants*, see below). Variables and constants have a *type*, which can be one of the five *intrinsic types*, or a *derived type*. Intrinsic types are part of the Fortran language. There are five different intrinsic types. In Fortran 90, there is additionally the possibility of defining derived types, which are defined by the user (see below). The five intrinsic types are:

Integer Type
For integer values (like 1, 2, 3, 0, -1, -2, …).

Real Type
For real numbers (such as 3.14, -100.876, 1.0 etc.). A processor must provide two different real types: The default real type and a type of higher precision, with the name double precision. Also this is a legacy of FORTRAN 77. Fortran 90 gives much more control over the precision of real and integer variables (through the kind specifier), see chapter on Numeric Precision, and there is therefore no need to use double precision. However, you will see double precision often used in older programs. For most of the exercises and examples in this manual the real type suffices. In the real word however, double precision is often required.

For now, if you prefer to use double precision in your programs, use:

```
real (kind=kind(1.0d0)) :: variable_name
```

Complex Type
For complex numbers. A complex value consists of two real numbers, the real part and the imaginary part. Thus, the complex number (2.0, -1.0) is equal to 2.0 – 1.0i.

Logical Type
There are only two logical values: .true. and .false. (note the dots around the words true and false).

Character Type
Data objects of the character type include characters and strings (a string is a sequence of characters). The length of the string can be specified by len (see the examples below). If no length is specified, it is 1.

Constants

A constant is a data object whose value cannot be changed.

A *literal constant* is a constant value without a name, such as 3.14 (a real constant), "Tanja" (a character constant), 300 (an integer constant), (3.0, -3.0) (a complex constant), .true. or .false. (logical constants. These two are the only logical constants available).

A *named constant* is a constant value with a name. Named constants and variables must be declared at the beginning of a program (or subprogram – see Chapter 4), in a so-called *type declaration statement*. The type declaration statement indicates the type and name of the variable or constant (note the two colons between the type and the name of the variable or constant). Named constants must be declared with the parameter attribute:

```
real, parameter :: pi = 3.1415927
```

Variables

Like named constants, variables must be declared at the beginning of a program (or subprogram) in a type declaration statement:

```
integer :: total
real :: average1, average2  ! this declares 2 real values
complex :: cx
logical :: done
```

```
character(len=80) :: line       ! a string consisting of 80 characters
```

These can be used in statements such as:

```
total = 6.7
average1 = average2
done = .true.
line = “this is a line”
```

Note that a character string is enclosed in double quotes (“).

Constants can be assigned trivially to the complex number cx:

```
cx = (1.0, 2.0)                 ! cx = 1.0 + 2.0i
```

If you need to assign variables to cx, you need to use cmplx:

```
cx = cmplx (1.0/2.0, -3.0)      ! cx = 0.5 – 3.0i
cx = cmplx (x, y)               ! cx = x + yi
```

The function cmplx is one of the intrinsic functions (see below).

Arrays

A series of variables of the same type can be collected in an *array*. Arrays can be one-dimensional (like vectors in mathematics), two-dimensional (comparable to matrices), up to 7-dimensional. Arrays are declared with the dimension attribute.

Examples:

```
real, dimension(5) :: vector          ! 1-dim. real array containing 5 elements
integer, dimension (3, 3) :: matrix   ! 2-dim. integer array
```

The individual elements of arrays can be *referenced* by specifying their *subscripts*. Thus, the first element of the array vector, vector(1), has a subscript of one. The array vector contains the real variables vector(1), vector(2), vector(3), vector(4), and vector(5). The array matrix contains the integer variables matrix(1,1), matrix(2,1), matrix(3,1), matrix(1,2), ..., matrix(3,3):

vector(1)	vector(2)	vector(3)	vector(4)	vector(5)

matrix(1,1)	matrix(1,2)	matrix(1,3)
matrix(2,1)	matrix(2,2)	matrix(2,3)
matrix(3,1)	matrix(3,2)	matrix(3,3)

The array vector could also have been declared with explicit lower bounds:

```
real, dimension (1:5) :: vector
```

All the following type declaration statements are legal:

```
real, dimension (-10:8) :: a1                ! 1-dim array with 19 elements
integer, dimension (-3:3, -20:0, 1:2, 6, 2, 5:6, 2) :: grid1    ! 7-dim array
```

The number of elements of the integer array grid1 is 7 x 21 x 2 x 6 x 2 x 2 x 2 = 14112.

You may not be able to use arrays with more than 7 dimensions. The standard requires that a compiler supports up to 7-dimensional arrays. A compiler may allow more than 7 dimensions, but it does not have to.

Character strings

The elements of a character string can be referenced individually or in groups.

With:

```
character (len=80) :: name
name = “Tanja”
```

Then

name(1:3) would yield the substring “Tan”

A single character must be referenced in a similar way:

name(2:2) yields the character “a”

If the lower subscript is omitted, it is assumed to be one, and if the upper subscript is omitted, it is supposed to be the length of the string.

Thus:

```
name (:3)     ! yields “Tan”
name (3:)     ! yields “nja”
name (:)      ! yields “Tanja”
```

Implicit typing

Fortran allows *implicit typing*, which means that variables do not have to be declared. If a variable is not declared, the first letter of its name determines its type: if the name of the variable starts with i, j, k, l, m, or n, it is considered to be an integer, in all other cases it is considered to be a real. However, it is good programming practice to declare all variables at the beginning of the program. The statement

```
implicit none
```

turns off implicit typing. All programs should start with this statement. (Implicit typing is not allowed in the F language).

Derived data types

We have seen that the Fortran language contains 5 intrinsic types (integer, real, complex, logical, and character). In addition to these, the user can define *derived types*, which can consist of data objects of different type.

An example of a derived data type:

```
type :: atom
   character (len=2) :: label
   real :: x, y, z
end type atom
```

This type can hold a 2-character atom name, as well as the atom's xyz coordinates.

An object of a derived data type is called a *structure*. A structure of type atom can be created in a type declaration statement like:

```
type(atom) :: carbon1
```

The components of the structure can be accessed using the *component selector* character (%):

```
carbon1%label = "C"
carbon1%x = 0.0000
carbon1%y = 1.3567
carbon1%z = 2.5000
```

Note that no spaces are allowed before and after the %!

One can also make arrays of a derived type:

```
type(atom), dimension (10) :: molecule
```

and use it like

```
molecule(1)%type = "C"
```

Arithmetic operators

The intrinsic arithmetic operators available in Fortran 90 are:

**	exponentiation
*	multiplication
/	division
+	addition
−	subtraction

These are grouped in order of precedence, thus, * has a higher precedence than +. The precedence can be overridden by using parentheses. For example:

```
3 * 2 + 1
```

yields 7, but

```
3 * (2+1)
```

yields 9.

For operators of equal strength the precedence is from left to right. For example:

```
a * b / c
```

In this statement, first a and b are multiplied, after which the results is divided by c. The exception to this rule is exponentiation:

```
2**2**3
```

is evaluated as 2**8, and not as 4**3.

Numeric expressions

An expression using any of the arithmetic operators (**, *, /, +, -), like the examples in the previous section, is called a *numeric expression*.

Be careful with integer divisions! The result of an integer division, *i.e.*, a division in which the numerator and denominator are both integers, is an integer, and may therefore have to be truncated. The direction of the truncation is towards zero (the result is the integer value equal or just less than the exact result):

3/2	yields 1
-3/2	yields –1
3**2	yields 9
3**(-2)	equals 1/3**2, which yields 0

Sometimes this is what you want. However, if you do not want the result to be truncated, you can use the real function. This function converts its argument to type real. Thus, real(2) yields a result of type real, with the value 2.0.

With the examples from above:

real(2)/3	yields 1.5
2/real(3)	yields 1.5
-2/real(3)	yields –1.5
real(3)**-2	yields 0.1111111111 (which is 1/9)

However:

real(2/3)	yields 0 (the integer division is performed first, yielding 0, which is then converted to a real.)

Note that the function real can have 2 arguments (see the table in the section on intrinsic functions, below). The second argument, an integer specifying the precision (kind value) of the result, is however optional. If not specified, the conversion will be to default precision. Kind values will be discussed later.

Numeric expressions can contain operands of different type (integer, real, complex). If this is the case, the type of the "weaker" operand will be first converted to the "stronger" type. (The order of the types, from strong to weak, is complex, real, integer.) The result will also be of the stronger type.

If we consider the examples above again, in

```
real(2)/3
```

The integer 3 is first converted to a real number 3.0 before the division is performed, and the result of the division is a real number (as we have seen).

Logical operators

The type logical can have only two different values: .true. and .false. (note the dots around the words true and false). Logical variables can be operated upon by logical operators. These are (listed in decreasing order of precedence):

==========

.not.
.and.
.or.
.eqv. and .neqv.

==========

The .and. is "exclusive": the result of a .and. b is .true. only if the expressions a and b are both true. The .or. is "inclusive": the result of a .or. b is .false. only if the expressions a and b are both false. Thus, if we have the logical constants:

```
logical, parameter :: on = .true.
logical, parameter :: off = .false.
```

Then:

```
.not. on            ! equals .false.
.not. off           ! equals .true.
on .and. on         ! equals .true.
on .and. off        ! equals .false.
off .and. off       ! equals .false.
on .or. on          ! equals .true.
on .or. off         ! equals .true.
off .or. off        !equals .false.
on .eqv. on         ! equals .true.
on .eqv. off        ! equals .false.
off .eqv. off       ! equals .true.
on .neqv. on        !equals .false.
on .neqv. off       ! equals .true.
off .neqv. off      ! equals .false.
```

Relational operators

Relational operators are operators that are placed between expressions and that compare the results of the expressions. The relational operators in Fortran 90 are:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
/=	not equal to

Logical expressions

Expressions that use the relational operators are *logical expressions*. The values of logical expressions can be either .true. or .false. Note that the range of numerical numbers in Fortran ranges from the largest negative number to the largest positive number (thus, -100.0 is smaller than 0.5).

A few examples:

```
real :: val1, val2
logical :: result

val1 = -3.5
val2 =  2.0

result = val1 < val2            ! result equals .true.
result = val1 >= val2           ! result equals .false.
result = val1 < (val2 – 2.0)    ! result equals .true.
```

Be careful though with comparing real numbers. Reals are represented with finite accuracy. Thus:

```
print*, 2 * 3.2
```

may give as output: 6.4000001. So instead of comparing two real variables a and b like:

```
a == b
```

it is safer to compare their difference:

```
real, parameter :: delta = 0.000001
abs(a-b) < delta       ! equals .true. if a and b are numerically identical
```

The function abs returns the absolute value of (a-b).

Intrinsic functions

Intrinsic functions are functions that are part of the language. Fortran, as a scientific language aimed at numerical applications, contains a large number of mathematical functions.

The mathematical functions are:

acos (x)	inverse cosine (arc cosine) function
asin (x)	inverse sine (arc sine) function
atan (x)	inverse tangent (arc tangent) function
atan2 (x)	arc tangent for complex numbers
cos (x)	cosine function
cosh (x)	hyperbolic cosine function
exp (x)	exponential function
log (x)	natural logarithm function
log10 (x)	common logarithm function
sin (x)	sine function
sinh (x)	hyperbolic sine function
sqrt (x)	square root function
tan (x)	tangent function
tanh (x)	hyperbolic tangent function

Some other useful intrinsic functions are:

abs (x)	absolute value of the numerical argument x
complx (x,y [, ikind])	convert to complex. The ikind argument is optional. If not specified, the conversion is to default precision.
floor (x)	greatest integer less than or equal to x. Examples: floor (3.4) equals 3, floor (-3.4) equals –4.
int (x)	convert to integer. If abs (x < 1), the result is 0. If abs (x) >= 1, the result is the largest integer not exceeding x (keeping the sign of x). Examples: int(0.3) equals 0, int (-0.3) equals 0, int (4.9) equals 4, int (-4.9) equals –4.
nint (x [, ikind])	rounds to nearest integer.
real (x [, ikind])	convert to real. The ikind argument is optional. If not specified, the conversion is to default precision.
mod (a,p)	remainder function. Result is a – int (a/p) * p. The arguments a and p must be of the same type (real or integer).
modulo (a,p)	modulo function. Result is a – floor (a/p) * p. The arguments a and p must be of the same type (real or integer).

Simple in- and output

As we have seen in the hello.f90 program above, characters can be displayed on the screen by the print* statement. Data can be transferred into the program by the read* statement. This is the simplest form of input/output (I/O), called *list-directed input/output*. As an example, with pi being declared as in the previous section, the statement

```
print*, "The number pi = ", pi
```

might appear on the screen as

```
The number pi = 3.141590
```

The exact format is dependent on the computer system used. Later we will see a more sophisticated form of I/O, using read and write, which gives the programmer more control over the format.

The following read statement (with x, y, and z being declared as variables of type real)

```
read*, x, y, z
```

expects three numbers to be typed, separated by a comma, one or more spaces, or a slash (/). The variable x will have the value of the first number typed, y will have the value of the second number typed, and z of the third number typed.

Comments

We have already seen the exclamation mark (!). All characters after the exclamation mark are ignored. Comments can be used for descriptive purposes, or for "commenting out" a line of code.

Continuation lines

The maximum length of a Fortran statement is 132 characters. Sometimes statements are so long that they don't fit on one line. The *continuation mark* (&), placed at the end of the line, allows the statement to continue on the next line. Fortran 90 allows a maximum of 39 continuation lines.

Thus, the following code

```
cos (alpha) = b*b + c*c –          &
   2*b*c*cos (gamma)
```

is identical to

```
cos (alpha) = b*b + c*c – 2*b*c*cos (gamma)
```

Summary

- A program starts with program program_name and ends with end program program_name.
- The first statement should always be implicit none.
- We have learned the different types of variables and constants in Fortran: integer, real, complex, logical and character, and how to declare them in type declaration statements.
- The arithmetic operators: **, *, /, + and -.
- The logical operators: .not., .and., .or., .eqv., and .neqv.
- The relational operators: <, <=, >, >=, == and /=.
- We learned the mathematical functions, and a selection of other intrinsic functions.
- We learned how to read in variables and how to write to the screen.

Exercises

1. Which of the following are valid names in Fortran 90:
   a. this_is_a_variable
   b. 3dim
   c. axis1
   d. y(x)
   e. dot.com
   f. DotCom
   g. z axis
2. Write a program that reads in a number, and computes the area of a circle that has a diameter of that size.
3. Find out, for your compiler, what the compiler flags are for displaying warning messages, and for issuing a warning when the compiler encounters non-standard source code.
4. Write a program that reads in a time in seconds, and computes how many hours and minutes it contains. Thus, 3700 should yield: 1 hour, 1 minute, and 40 seconds. (Hint: use the mod function).

3. Control Constructs

Control Constructs

A program can consist of several statements, which are executed one after the other:

```
program program_name
   implicit none
   statement1
   statement2
   statement3
   statement4
end program_name
```

However, this rigid sequence may not suit the formulation of the problem very well. For example, one may need to execute the same group of statements many times, or two different parts of the program may need to be executed, depending on the value of a variable. For this, Fortran 90 has several constructs that alter the *flow* through the statements. These include *if constructs*, *do loops*, and *case constructs*.

If constructs:

The simplest form of an if construct is:

```
if (logical expression) then
   statement
end if
```

as in:

```
if (x < y) then
   x = y
end if
```

The statement x = y is only executed if x is smaller than y.

Several statements may appear after the logical expression. The if block can also be given a name, so the more general form of the if construct is:

```
[name:] if (logical expression) then
   ! various statements
      . . .
end if [name]
```

Both endif and end if are allowed. The name preceding the if is optional (but if it is specified, the name after the endif must be the same).

The block if statement can also contain an *else block*:

```
[name:]  if (logical expression) then
            ! various statements
            . . .
         else
            ! some more statements
            . . .
         end if [name]
```

Block if statements can be nested:

```
[name:]  if (logical expression 1) then
            ! block 1
         else if (logical expression 2) then
            ! block 2
         else if (logical expression 3) then
            ! block 3
         else
            ! block 4
         end if [name]
```

Example (try to follow the logic in this example):

```
if ( optimisation ) then
   print*, "Geometry optimisation: "
   if ( converged ) then
      print*, "Converged energy is ", energy
   else
      print*, "Energy not converged. Last value: ", energy
   end if
else if (singlepoint ) then
   print*, "Single point calculation: "
   print*, "Energy is ", energy
else
   print*, "No energy calculated."
end if
```

Indentation is optional, but highly recommended: a consistent indentation style helps to keep track of which if, else if, and end if belong together (*i.e.*, have the same "if level").

Do loops

A program often needs to repeat the same statement many times. For example, you may need to sum all elements of an array.

You could write:

```
real, dimension (5) :: array1
real :: sum

! here some code that fills array1 with numbers
. . .

sum = array1(1)
sum = sum + array1(2)
sum = sum + array1(3)
sum = sum + array1(4)
sum = sum + array1(5)
```

But that gets obviously very tedious to write, particularly if array1 has many elements. Additionally, you may not know beforehand how many times the statement or statements need to be executed. Thus, Fortran has a programming structure, the do loop, which enables a statement, or a series of statements, to be carried out iteratively.

For the above problem, the do loop would take the following form:

```
real, dimension (5) :: array1
real :: sum
integer :: i        ! i is the “control variable” or counter

! here some code that fills array1 with numbers
. . .

sum = 0.0    ! sum needs to be initialised to zero
do i = 1, 5
   sum = sum + array1(i)
end do
```

Both enddo and end do are allowed.

It is possible to specify a name for the do loop, like in the next example. This loop prints the odd elements of array2 to the screen. The name (print_odd_nums in this case) is optional. The increment 2 specifies that the counter i is incremented with steps of 2, and therefore, only the odd elements are printed to the screen. If no increment is specified, it is 1.

```
real, dimension (100) :: array2
integer :: i

! here some code that fills array2 with numbers
. . .

print_odd_nums:   do i = 1, 100, 2
                      print*, array2(i)
                  end do print_odd_nums
```

Do loops can be nested (one do loop can contain another one), as in the following example:

```
real, dimension (10,10) :: a, b, c   ! matrices
integer :: i, j, k

! here some code to fill the matrices a and b
. . .

! now perform matrix multiplication: c = a + b
do i = 1, 10
   do j = 1, 10
      c(i, j) = 0.0
      do k = 1, 10
         c(i, j) = c(i, j) + a(i, k) + b(k, j)
      end do
   end do
end do
```

Note the indentation, which makes the code more readable.

Endless Do

The endless do loop takes the following form:

```
[doname:]   do
               ! various statements
               exit [doname]
               ! more statements
            end do [doname]
```

Note the absence of the control variable (counter). As before, the name of the do loop is optional (as indicated by the square brackets).

To prevent the loop from being really "endless", an exit statement is needed. 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.

The exit statement usually takes the form

```
if (expression) then
   exit
end if
```

as in the following example:

```
program integer_sum
! this program sums a series of numbers given by the user
! example of the use of the endless do construct
    implicit none
    integer :: number, sum

    sum = 0
    do
       print*, “give a number (type –1 to exit): “
       read*, number

       if (number == -1) then
          exit
       end if

       sum = sum + number
    end do

    print*, “The sum of the integers is “, sum

    end program integer_sum
```

The name of the do loop can be specified in the exit statement. This is useful if you want to exit a loop in a nested do loop construct:

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

          if (k==2) then
             exit jloop
      end do kloop

    end do jloop
end do iloop
```

When the exit statement is executed, the program continues at the next executable statement after end do jloop. Thus, the first time that exit is reached is when i=1, j=1, k=2, and the program continues with i=2, j=1, k=1.

A statement related to exit is the cycle statement. If a cycle statement is executed, the program continues at the start of the next iteration (if there are still iterations left to be done).

Example:

```
program cycle_example
   implicit none
   character (len=1) :: answer
   integer :: i
   do i = 1, 10
      print*, “print i (y or n)?”
      read*, answer
      if (answer == “n”) then
         cycle
      end if
      print*, i
   end do
end program cycle_example
```

Case constructs

The case construct has the following form:

```
[name:]          select case (expression)
                    case (selector1)
                       ! some statements
                       . . .
                    case (selector2)
                       ! other statements
                       . . .
                    case default
                      ! more statements
                       . . .
                 end select [name]
```

As usual, the name is optional. The value of the selector, which can be a logical, character, or integer (but not real) expression, determines which statements are executed. The case default block is executed if the expression in select case (expression) does not match any of the selectors.

A range may be specified for the selector, by specifying an lower and upper limit separated by a colon:

```
case (low:high)
```

Example:

```
select case (number)
   case ( : -1)
      print*, “number is negative”
   case (0)
      print*, “number is zero”
   case (1 : )
      print*, “number is positive”
end select
```

The following example program asks the user to enter a number between 1 and 3. The print and read are in an endless loop, which is exited when a number between 1 and 3 has been entered.

```
program case_example
   implicit none
   integer :: n
   ! Ask for a number until 1, 2, or 3 has been entered
   endless: do
      print*, "Enter a number between 1 and 3: "
      read*, n
      select case (n)
      case (1)
         print*, "You entered 1"
         exit endless
      case (2)
         print*, "You entered 2"
         exit endless
      case (3)
         print*, "You entered 3"
         exit endless
      case default
         print*, "Number is not between 1 and 3"
      end select
   end do endless
end program case_example
```

Summary

In this chapter we learned the following control constructs:

- block if statements.
- do loops (including endless do loops).
- case statements.

Exercises

5. Write a program which calculates the roots of the quadratic equation $ax^2 + bx + c = 0$. Distinguish between the three cases for which the discriminant ($b^2 - 4ac$) is positive, negative, or equal to zero. Use an if construct. You will also need to use the intrinsic function cmplx.
6. Consider the Fibonacci series:
   1 1 2 3 5 8 13 …
   Each number in the series (except the first two, which are 1) is the sum from the two previous numbers. Write a program that reads in an integer limit, and which prints the first limit terms of the series. Use an nested if block structure. (You need to distinguish between several cases: limit < 0, limit =1, etc.)
7. Rewrite the previous program using a case construct.
8. Write a program that
   defines an integer array to have 10 elements
   a) fills the array with ten numbers
   b) reads in 2 numbers (in the range 1-10)
   c) reverses the order of the array elements in the range specified by the two numbers.
   Try not to use an additional array.
9. Write a program that reads in a series of integer numbers, and determines how many positive odd numbers are among them. Numbers are read until a negative integer is given. Use cycle and exit.

4. Procedures

Program units

A program can be built up from a collection of program units (the main program, modules and external subprograms or procedures). Each program must contain one (and only one) main program, which has the familiar form:

```
program program_name
   implicit none
   ! type declaration statements
   ! executable statements
end program program_name
```

Modules will be discussed later.

Procedures

A subprogram or procedure is a computation that can be "called" (invoked) from the program. Procedures generally perform a well-defined task. They can be either functions or subroutines. Information (data) is passed between the main program and procedures via *arguments*. (Another way of passing information is via modules, see Chapter 6.) A function *returns* a single quantity (of any type, including array), and should, in principle, not modify any of its arguments. (In the stricter F language, a function is simply not allowed to modify its arguments). The quantity that is returned is the *function value* (having the name of the function). We have already seen one type of functions in Chapter 2, namely built-in or intrinsic functions, which are part of the Fortran 90 language (such as cos or sqrt).

An example of a function:

```
function circle_area (r)
! this function computes the area of a circle with radius r
   implicit none
   ! function result
   real :: circle_area
   ! dummy arguments
   real :: r
   ! local variables
   real :: pi
   pi = 4 * atan (1.0)
   circle_area = pi * r**2
end function circle_area
```

The structure of a procedure closely resembles that of the main program. Note also the use of implicit none. Even if you have specified implicit none in the main program, you need to specify it again in the procedure.

The r in the function circle_area is a so-called *dummy argument*. Dummy arguments are replaced by *actual arguments* when the procedure is called during execution of the program. Note that the function has a "dummy arguments" block and a "local variables" block, separated by comments. While this is not required, it makes the program clearer.

The function can be used in a statement like:

```
a = circle_area (2.0)
```

This causes the variable a to be assigned the value 4π.

This is, by the way, not a very efficient way to calculate the area of a circle, as π is recalculated each time this function is called. So if the function needs to be called many times, it will be better to obtain π differently, for example by declaring:

```
real, parameter :: pi = 3.141592654
```

The result of a function can be given a different name than the function name by the result option:

```
function circle_area (r) result (area)
! this function computes the area of a circle with radius r
   implicit none
   ! function result
   real :: area
   ! dummy arguments
   real :: r
   ! local variables
   real, parameter :: pi = 3.141592654
   area = pi * r**2
end function circle_area
```

The name specified after result (area in this case) *must* be different from the function name (circle_area). Also note the type declaration for the function result (area). This function is used in the same way as before:

```
a = circle_area (radius)
```

The result option is in most cases optional, but it is required for recursive functions, *i.e.*, functions that call themselves (see paragraph on "recursive functions" below).

An example of a subroutine:

```
subroutine swap (x, y)
   implicit none
   ! dummy arguments
   real :: x, y
   ! local variables
   real :: buffer
   buffer = x              ! store value of x in buffer
   x = y
   y = buffer
end subroutine swap
```

A subroutine is different from a function in several ways. Subroutines can modify their arguments, and they do not return a single "result" as functions do. Functions return a value that can be used directly in expressions, such as:

```
a = circle_area (radius)
```

A subroutine must be "call"ed, as in:

```
call swap (x,y)
```

The general rule is that it is best to use a function if the procedure computes only one result, and does not much else. In all other cases, use a procedure.

External procedures

An example of a program that contains two functions:

```
program angv1v2

   implicit none

   real, dimension (3) :: v1, v2
   real :: ang

   ! define two vectors v1 and v2

   v1(1) = 1.0
   v1(2) = 0.0
   v1(3) = 2.0

   v2(1) = 1.5
   v2(2) = 3.7
   v2(3) = 2.0

   print*, "angle  = ", ang (v1, v2)

end program angv1v2

! ang computes the angle between 2 vectors vect1 and vect2
function ang (vect1, vect2 )

   implicit none

   ! function result
   real :: ang

   ! dummy arguments
   real, dimension (3), intent (in) :: vect1, vect2

   ! local variables
   real :: cosang, norm

   cosang = vect1(1)*vect2(1) + vect1(2)*vect2(2) + vect1(3)*vect2(3)
   cosang = cosang / (norm(vect1)*norm(vect2))
   ang = acos (cosang)

end function ang

! norm returns the norm of the vector v
function norm (v)

   implicit none

   real :: norm

   ! dummy arguments
   real, dimension (3) :: v

   norm = sqrt ( v(1)**2 + v(2)**2 + v(3)**2)

end function norm
```

This program illustrates that the actual argument (v1 and v2) may have a name different from the dummy arguments vect1 and vect2 of the function ang. This allows the same function to be called with different arguments. The intent attribute is explained in the next paragraph. Note that the type of the function norm must be declared in the function ang. An alternative to this is to provide an explicit interface, see section on Interfaces below.

As mentioned before, a subroutine must be "call"ed, as the following program illustrates:

```
program swap_xy

   implicit none
   real :: x, y

   x = 1.5
   y = 3.4
   print*, "x = ", x, " y = ", y
   call swap (x,y)
   print*, "x = ", x, " y = ", y

end program swap_xy

subroutine swap (x, y)

   implicit none

   ! dummy arguments
   real, intent (inout) :: x, y

   ! local variables
   real :: buffer

   buffer = x                ! store value of x in buffer
   x = y
   y = buffer
end subroutine swap
```

When executed, this program gives as output:

x = 1.5 y = 3.4
x = 3.4 y = 1.5

(The exact format of the displayed numbers is dependent on the computer system used.)

Note that the functions appear after the end of the main program. Subroutines or functions that are not contained in the main program are called *external procedures*. These are the most common type of procedures. External procedures are stand-alone procedures, which may be developed and compiled independently of other procedures and program units. The subroutines do not have to be in the same file as the main program. If the two functions in the example program angv1v2 above are in a file vector.f90, and the main program is in a file called angle.f90, then a compilation of the program may look like this (for the SGI MIPS Fortran 90 compiler for example. The actual format of the compilation is compiler-specific, and may look different with another compiler):

```
f90 –ansi –fullwarn –o angle angle.f90 vector.f90
```

The compiler flag –ansi causes the compiler to generate messages when it encounters source code that does not conform to the Fortran 90 standard, -fullwarn turns on all warning messages, and –o specifies the name of the output file, to which the executable will be written to.

The compilation can also be done in two steps:

```
f90 –ansi –fullwarn –c angle.f90 vector.f90
f90 –ansi –fullwarn –o angle angle.o vector.o
```

The first step creates binary object files vector.o and angle.o. The second (link) step creates the executable (angle).

Intent

Fortran allows the specification of the "intention" with which arguments are used in the procedure:

intent (in): Arguments are only used, and not changed
intent (out): Arguments are overwritten
intent (inout): Arguments are used *and* overwritten

Consider the following example:

```
subroutine intent_example (a, b, c)

   implicit none

   ! dummy arguments
   real, intent (in) :: a
   real, intent (out) :: b
   real, intent (inout) :: c

   b = 2 * a
   c = c + a * 2.0

end subroutine intent_example
```

In this subroutine, a is not modified, and thus has intent (in); b is given a value and has therefore intent (out); c is used and modified, intent (inout). It is good programming practice to use intent. Firstly, it makes procedures more transparent, *i.e.*, it is clearer what the procedure does. Secondly, the compiler may catch programming mistakes, because most compilers will warn you if you, for example, try to modify an argument that has intent (in). Thirdly, it may help optimisation if the compiler knows which arguments are changed in a subroutine.

Even though it is advisable to use intent, it is possible to introduce bugs in the program by giving arguments the wrong intent. Consider the following code:

```
program test
   implicit none
   integer, dimension (10) :: array
   integer :: i
   do i = 1, 10
      array(i) = i
   end do
   call modify_array (a)
end program test
subroutine modify_array (a)
   implicit none
   ! dummy arguments
   integer, dimension (10), intent (inout) :: a
   ! local variables
   integer :: i
   do i = 1,3
      a(i) = 0.0
  end do
end subroutine modify_array
```

The intent of array a in the subroutine has to be inout, even though it seems like you are only writing into the array, and do not need to know the values of its elements. If you would make the intent out however, then it is possible that, after calling the subroutine, the elements a(4) to a(10) contain "garbage" (unpredictable contents), because the subroutine did not read in these elements (so cannot know their values), but did write them out (thereby overwriting their previous values).

Interfaces

The interface of a procedure is a collection of the names and properties of the procedure and its arguments. When a procedure is external, the compiler will (in most cases) not know about its interface, and cannot check if the procedure call is consistent with the procedure declaration (for example, if the number and types of the arguments match).

Providing an *explicit interface* makes such cross-checking possible. It is thus good programming practice to specify interfaces for external procedures. In certain cases, an interface is required. So it is best to provide an explicit interface for external procedures. (For space-saving reasons, the example programs in this manual do not always have them).

The interface block contains the name of the procedure, and the names and attributes (properties) of all dummy arguments (and the properties of the result, if it defines a function).

If we take as example the program swap_xy from above, then the interface for the subroutine swap would look like:

```
interface
   subroutine swap (x, y)
      real, intent (inout) :: x, y
   end subroutine swap
end interface
```

The interface block is placed at the beginning of the program (or at the beginning of a procedure), together with the declaration statements:

```
program swap_xy

   implicit none

   ! local variables
   real :: x, y

   ! external procedures
   interface
      subroutine swap (x, y)
         real, intent (inout) :: x, y
      end subroutine swap
   end interface

   x = 1.5
   y = 3.4
   print*, "x = ", x, " y = ", y
   call swap (x,y)
   print*, "x = ", x, " y = ", y

end program swap_xy

subroutine swap (x, y)

   implicit none

   ! dummy arguments
   real, intent (inout) :: x, y

   ! local variables
   real :: buffer

   buffer = x                ! store value of x in buffer
   x = y
   y = buffer

end subroutine swap
```

If a procedure proc1 calls another procedure proc2, then the interface block of proc2 should be placed at the beginning of the procedure proc1.

Another way of providing explicit interfaces will be discussed in the chapter on Modules (Chapter 6).

Recursive procedures

A procedure that calls itself (directly or indirectly) is called a recursive procedure. Its declaration must be preceded by the word recursive. When a function is used recursively, the result option must be used.

The factorial of a number (n!) can be computed using a recursive function:

```
recursive function nfactorial (n) result (fac)
! computes the factorial of n (n!)

   implicit none

   ! function result
   integer :: fac

   ! dummy arguments
   integer, intent (in) :: n

   select case (n)
      case (0:1)
         fac = 1
      case default
         fac = n * nfactorial (n-1)
   end select

end function nfactorial
```

Internal procedures

Internal procedures are contained within a program unit. A main program containing internal procedures has the following form:

```
program program_name
   implicit none
      ! type declaration statements
      ! executable statements
    . . .
   contains
      ! internal procedures
    . . .
end program program_name
```

With our function circle_area from above:

```
program area
   implicit none
   real :: radius, a
   radius = 1.4
   a = circle_area (radius)
   print*, “The area of a circle with radius “, radius, “ is “, a
contains
   function circle_area (r)
   ! this function computes the area of a circle with radius r
      implicit none
      ! function result
      real :: circle_area
      ! dummy arguments
      real, intent (in) :: r
      ! local variables
      real, parameter :: pi = 3.141592654
      circle_area = pi * r**2
   end function circle_area
end program area
```

An internal procedure is local to its *host* (the program unit containing the internal procedure), and the environment *(i.e.*, the variables and other declarations) of the host program is known to the internal procedure.

Thus, the function circle_area knows the value of the variable radius, and we could have written the function also like:

```
   function circle_area
   ! this function computes the area of a circle with radius r
      implicit none
      ! function result
      real :: circle_area
      ! local variables
      real, parameter :: pi = 3.141592654
      circle_area = pi * radius**2
   end function circle_area
end program area
```

and simple called the function like:

```
a = circle_area
```

However, the first form allows the function to be called with varying arguments (and it more transparent as well):

```
a1 = circle_area (radius)
a2 = circle_area (3.5)
```

Internal procedures are not very common. In most cases, it is better to use external procedures. External procedures can be called from more than one program unit, and they are safer: The variables of the calling program are hidden from the procedure, *i.e.*, the procedure does not know the values of the variables (unless they are passed as arguments), and it can only change them if they are passed as arguments with intent (inout). However, an advantage of internal procedures is that they can be better optimised by the compiler.

Assumed character length

A character dummy argument can be declared with an asterisk for the length the len parameter. This allows the procedure to be called with character strings of any length. The length of the dummy argument is taken from that of the actual argument.

Example:

```
program assumed_char
   implicit none
   character (len=5) :: name
   name = “Tanja”
   call print_string (name)
end program assumed_char
subroutine print_string (name)
   implicit none
   ! dummy arguments
   character (len=*), intent (in) :: name
   print*, name
end subroutine print_string
```

Summary

This chapter discussed how to break a program down into manageable units, which each correspond to a specific programming task. The program units we saw in this chapter are:

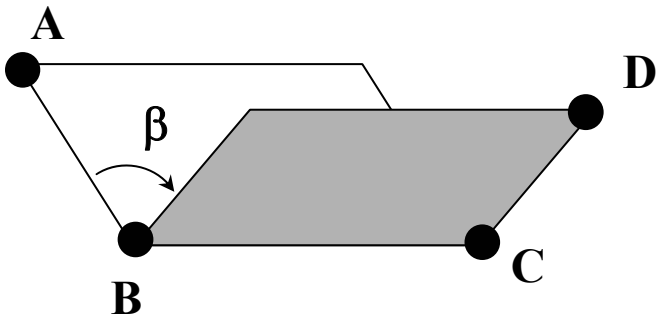
- The main program
- External procedures (subroutines and functions)
- Internal procedures (subroutines and functions)
- Recursive procedures

Good programming practice requires the use of the intent attribute for the dummy arguments of procedures, and the use of interface blocks for external procedures.

Exercises

Choose any two of exercises 11-13:

10. Write a program that, given the xyz coordinates of four atoms, returns the dihedral between the atoms. The dihedral (or torsion) angle of four atoms A-B-C-D is defined as the angle between the plane containing the atoms A, B, and C, and the plane containing B, C, and D.



11. Consider the Fibonacci series:
    1 1 2 3 5 8 13 …
    Each number in the series (exceptt the first 2, which are 1) is the sum from the two previous numbers. Write a program that computes the *n*th number in the Fibonacci series, where *n* is given by the user. Use a recursive function.

12. Bubble sort.
    Create an unordered data set of integers (for example by reading them in), and write a subroutine to sort them in ascending order. One of the easiest sort algorithms is bubble sort. Start at the lower end of the array. Compare elements 1 and 2, and swap them if necessary. Then proceed to the next 2 elements (2 and 3), and continue this process through the entire array. Repeat the whole process ndim (dimension of array) times. Note that in successive rounds you only have to go through smaller and smaller sections of the array, because the last element(s) should now already be sorted. This process is called "bubble sort" because the larger elements appear to bubble through the array to reach their places.

5 More on Arrays

Declaring arrays

In chapter 2 we have seen that we can declare arrays with the dimension attribute:

```
real, dimension (3) :: coords          ! 1-dimensional real array
integer, dimension (10,10) :: block    ! 2-dimensional integer array
```

Up to seven dimensions are allowed.

An alternative way of declaring these arrays is as follows:

```
real :: coords(3)
integer :: block(10,10)
```

In this manual, the first method is used.

Arrays can also be declared with explicit lower bounds. For the above arrays:

```
real, dimension (1:3) :: coords
integer, dimension (-5:5, 0:9) :: block
```

The type declaration statement for the array block shows that the lower bound does not have to be 1. If the lower bound is not specified explicitly, it is taken to be 1.

Array terminology

Rank: The rank of an array is the number of dimensions it has. In our examples above, the rank of coords is 1 and the rank of block is 2.

Extent: The number of elements along a dimension is its extent. Thus, the extent of coords is 3 and the extent of both the first and second dimension of block is 10.

Shape: The shape of an array is a one-dimensional integer array, containing the number of elements (the extent) in each dimension. Thus, the shape of array coords is (3), and the shape of block is (10,10). Two arrays of the same shape are "conformable".

Size: The size of an array is the number of elements it contains. The size of coords is 3 and the size of block is 100.

Array assignment statements

Array elements can be given a value in the usual way:

```
coords(1) = 3.5
block(3,2) = 7
```

Or in a loop:

```
do i = 1,3
   coords(i) = 0.0
end do
```

For arrays of rank one (one-dimensional arrays), the following shorthand notation is also possible:

```
coords = (/ 1.3, 5.6, 0.0 /)
```

These shorthand notations of "constructing" the array elements are called *array constructors*. Note that the "(/" and "/)" are a single symbol, thus, no spaces are allowed between the (and / characters.

The following constructors are also allowed:

```
coords = (/ (2.0*i, i = 1, 3) /)        ! yields (2.0, 4.0, 6.0)
odd_ints = (/ (i, i = 1, 10, 2) /)      ! yields (1, 3, 5, 7, 9)
```

These two examples use so-called *implied do loops*. Note the additional parentheses around the implied do loop.

The array constructors allow the definition of array constants:

```
integer, dimension (8), parameter :: primes = (/ 1, 2, 3, 7, 11, 13, 17, 19 /)
```

Array sections

Sections of arrays can be referenced. For example, consider an integer array a with dimension (3,4).

a(1,1)	a(1,2)	a(1,3)	a(1,4)
a(2,1)	a(2,2)	a(2,3)	a(2,4)
a(3,1)	a(3,2)	a(3,3)	a(3,4)

Then, a(1:2, 3) references the elements a(1,3) and a(2,3). The whole last column can be referenced as a(1:3,4) or simply a(:, 4), and the first row as a(1, :). Optionally, a stride can be specified as well. The syntax is (for each dimension):

[lower] : [upper] [: stride]

In a(1, 1:4:2) lower= 1, upper = 4 and stride = 2 (for the second dimension). Thus, a(1, 1:4:2) references the elements a(1,1) and a(1,3).

Array expressions

The arithmetic operators (**, *, /, +, -) can be applied to arrays (or array sections) that have the same shape (are conformable).

For example, a two-dimensional array b(2,3) can be added to the array section a(2:3, 1:3) of the array a of the previous section. If the array c is an array of dimension (2,3), then the expression

```
c = a(2:3,1:3) + b
```

causes the elements of the array c to have the following values:

```
c(1,1) = a(2,1) + b(1,1)
c(2,1) = a(3,1) + b(2,1)
c(1,2) = a(2,2) + b(1,2)
c(2,2) = a(3,2) + b(2,2)
c(1,3) = a(2,3) + b(1,3)
c(2,3) = a(3,3) + b(2,3)
```

The same can be achieved by using a do loop:

```
do i = 1, 3
   do j = 1, 2
      c(j,i) = a(j+1,i) + b(j,i)
   end do
end do
```

But the expression `c = a(2:3,1:3) + b` is clearly more concise.

The operator + in the above example can be replaced by any of the other arithmetic operators. The following expressions are all valid:

```
c = a(2:3,1:3) * b          ! c(1,1) = a(2,1) * b(1,1), etc.
c = a(2:3,1:3) / b          ! c(1,1) = a(2,1) / b(1,1), etc.
c = a(2:3,1:3) – b          ! c(1,1) = a(2,1) - b(1,1), etc.
c = a(2:3,1:3) ** b         ! c(1,1) = a(2,1) ** b(1,1), etc.
```

Note that the operation is done element by element. Thus, the result of the multiplication `a(2:3,1:3) * b` is not a matrix product!

Scalars can be used in an array expression as well. Thus, `a / 2.0` has the effect of dividing all elements of the array `a` by 2.0, and `a**2` raises all the elements of `a` to the power 2.

Array expressions can be used to avoid do-loops. For example, the expression

```
a(1:4) = a(2:5)
```

Is equivalent to

```
do i = 1, 4
   a(i) = a(i+1)
end do
```

However, if the do loop iterations are interdependent, then the do loop and the array expression are not equivalent. This is because in the do loop the elements of the array are updated after each iteration, and in the array expression the updating is done only after all the elements have been processed.

For example, if the array `a` contains the numbers 1, 2, 3, 4, 5, then

```
do i = 1,4
   a(i+1) = a(i)
end do
```

yields 1, 1, 1, 1, 1

(After the first iteration, the array contains 1, 1, 3, 4, 5, after the second 1, 1, 1, 4, 5, after the third 1, 1, 1, 1, 5 and after the fourth 1, 1, 1, 1, 1).

However,

```
a(2:5) = a(1:4)
```

yields 1, 1, 2, 3, 4.

Using array syntax instead of do loops may help an optimising compiler to optimise the code better in specialised cases (for example on a parallel machine).

Dynamic arrays

Sometimes you don't know the necessary size of an array until the program is run, the size may for example depend on some calculations, or be defined by the user. In Fortran 90, this can be done by using *dynamic arrays*. A dynamic array is an array of which the size is not known at compile time (the rank must be specified, however), but becomes known when running the program.

A dynamic array has to be declared with the attribute allocatable, and it has to be "allocated" when its size is known and before it is used.

```
program dynamic_array

   implicit none

   real, dimension (:,:), allocatable :: a
   integer :: dim1, dim2
   integer :: i, j

   print*, "Give dimensions dim1 and dim2: "
   read*, dim1, dim2

   ! now that the size of a is know, allocate memory for it
   allocate ( a(dim1,dim2) )

   do  i = 1, dim2
       do j = 1, dim1
           a(j,i) = i*j
           print*, "a(",j,",",i,") = ", a(j,i)
       end do
   end do

   deallocate (a)
end program dynamic_array
```

When the array is no longer needed, it should be deallocated. This frees up the storage space used for the array for other use.

Assumed shape arrays

Arrays can be passed as arguments to procedures, as the following example illustrates:

```
program dummy_array
   implicit  none
   integer, dimension (10) :: a
   call fill_array (a)
   print, a
end program dummy_array
subroutine fill_array (a)
   implicit none
   ! dummy arguments
   integer, dimension (10), intent (out) :: a
   ! local variables
   integer :: i
   do i = 1, 10
      a(i) = i
   end do
end subroutine fill_array
```

However, written like this, the subroutine fill_array can only be called with arrays of dimension 10. It would clearly be advantageous if routines can be used for arrays of any size. To accomplish this, several techniques can be used to "hide" the array size from the procedure. In Fortran 90, the most flexible way of doing this is by using the *assumed shape* technique. In the procedure, the shape of the array is not specified, but is taken automatically to be that of the corresponding actual argument. The size of the array (the number of elements it contains, size_a in the example below), is determined in the subroutine using the intrinsic function size. size (array, dim) returns the number of elements along a specified dimension dim. The argument dim is optional. If it not specified, size sums the number of elements in each dimension.

The above program can be rewritten as follows:

```
program dummy_array
   implicit none
   integer, dimension (10) :: a
   interface
      subroutine fill_array (a)
         integer, dimension ( : ) intent (out) :: a
         integer :: i
      end subroutine fill_array
   end interface
   call fill_array (a)
   print, a
end program dummy_array
subroutine fill_array (a)
   implicit none
   ! dummy arguments
   integer, dimension ( : ), intent (out) :: a
   ! local variables
   integer :: i, size_a
   size_a = size (a)
   do i = 1, size_a
      a(i) = i
   end do
end subroutine fill_array
```

Note that, in this case, the explicit interface is required. The procedure fill_array can now be called with arrays of any size.

Multidimensional arrays can also be passed to procedures in this way:

```
program example_assumed_shape
   implicit none
   real, dimension (2, 3) :: a
   integer :: i, j
   interface
      subroutine print_array (a)
         real, dimension ( : , : ), intent (in) :: a
      end subroutine print_array
   end interface
   do i = 1, 2
      do j = 1, 3
         a(i, j) = i * j
      end do
   end do
   call print_array (a)
end program example_assumed_size
subroutine print_array
   implicit none
   ! dummy arguments
   real,dimension ( : , : ), intent (in) :: a
   print*, a
end subroutine print_array
```

Note that the rank (*i.e.*, the number of dimensions) must be explicitly defined in the procedure, but the shape (the extent of each dimension) is taken from the actual argument. If required, in the subroutine print_array the size of the array can be found out by the function size. In this case, size (a) would yield 6, size (a, 1) would yield 2, and size (a, 2) would yield 3.

Summary

This chapter showed some more advanced array manipulations, such as implied do loops and array expressions. A very useful concept is that of dynamic (allocatable) arrays, which did not exist in FORTRAN 77. We have seen how the shape of an array can be passed implicitly to a procedure by using assumed shape arrays.

Exercises

13. Given the array declaration:
    real, dimension (24, 10) :: a

    Write array sections representing:
    a) the second column of a
    b) the last row of a
    c) the block in the upper left corner of size 2 x 2.

14. Rewrite exercise 3.8 using array syntax instead of do loops.

6. Modules

Modules

A module serves as a packaging means for subprograms, data and interface blocks. Modules are a new and powerful feature of Fortran 90. They make common blocks (routinely used in FORTRAN 77) and include statements obsolete.

A module consists of two parts: a specification part for the declaration statements (including interface blocks and type and parameter declarations), and a subprogram part.

The general form of a module is:

```
module module_name
   ! specification statements
contains
   ! procedures
end module module_name
```

Modules can contain just the specification part or the subprogram part, or both.

The following example contains both:

```
module constants
implicit none
real, parameter :: pi = 3.1415926536
real, parameter ::  e = 2.7182818285
contains
   subroutine show_consts()
      print*, “pi = “, pi
      print*,  e = “. e
   end subroutine show_consts
end module constants
```

Note the implicit none. Just like with procedures, it is good programming practice to use implicit none in modules. It only needs to be specified once *i.e.*, it is not necessary to specify implicit none again in the module's procedures.

Modules are accessed by the use statement:

```
program module_example

   use constants
   implicit none

   real :: twopi

   twopi = 2 * pi
   call show_consts()
   print*, "twopi = ", twopi

end program module_example
```

The use statement makes available to the main program all the code (specification statements and subprograms) in the module constants. It supplies an explicit interface of all the module's procedures. The use statement has to be the first statement in the program (it comes even before implicit none), only comments are allowed before use.

When a subroutine is defined in a module, then there is no need to provide an explicit interface in the calling program (as long as the module's contents are made available to the program via the use statement).

Accessibility

By default, everything in a module is publicly available, that is, the use statement in the main program makes available all of the code in the module. However, accessibility can be controlled by the private and public attributes. Everything that is declared private is not available outside the module.

Example:

```
module convertT
   implicit none
   real, parameter, private :: factor = 0.555555556
   integer, parameter, private :: offset = 32
   contains
      function CtoF (TinC) result (TinF)
         ! funtion result
         real :: TinF
         ! dummy argument
        real, intent (in) :: TinC
         TinF = (TinC/factor) + offset
      end function CtoF
      function FtoC (TinF) result (TinC)
         !function result
         real :: TinC
         ! dummy argument
         real :: TinF
        TinC = (TinF-offset) * factor
      end function FtoC
end module convertT
```

The following program uses the module convertT:

```
program convert_temperature
   use convertT
   implicit none
   print*, "20 Celcius = ", CtoF (20.0), " Fahrenheit"
   print*, "100 Fahrenheit = ", FtoC (100.0), " Celcius"
end program convert_temperature
```

The module defines two constants, factor and offset, which are not available to the program convert_temperature. The functions CtoF and FtoC, however, are available to the program. Thus, the statement:

```
print*, offset
```

would induce an error message from the compiler. (For example, my Intel Fortran Compiler would give the error message: "In program unit CONVERT_TEMPERATURE variable OFFSET has not been given a type").

For data objects, like factor and offset in the example above, the public and private attributes occur in the type declaration statement:

```
real, private :: factor
integer, private :: offset
```

For procedures, they must be defined in public and private statements:

```
public :: CtoF, FtoC
```

The default accessibility of the module can be set by the public or private statements. If private is specified, then all module contents are private, except those that are explicitly defined as public:

```
module convertT
private
public :: CtoF, FtoC
```

Selecting module elements

Generally, the use statement makes available all (public) elements of a module. However, when only a subset of the module is needed, the accessibility can be restricted with only.

The example of the previous section could have be written in the following way:

```
module convertT
   public

end module convertT

program convert_temperature
   use convertT, only: CtoF, FtoC

end program convert_temperature
```

The only option safeguards the module elements that are not needed by making them inaccessible to the program. It can also make programs more transparent, by showing the origin of data objects or procedures, particularly if the program uses several modules:

```
program only_example
   use module1, only: sphere, triangle
   use module2, only: compute_gradient
   use module3, only: element1, element2, element3


end program only_example
```

It is now obvious that the procedure compute_gradient is defined in module2.

Data encapsulation

Modules allow data and operations to be hidden from the rest of the program. *Data encapsulation* refers to the process of hiding data within an "object", and allowing access to this data only through special procedures, called *member functions* or *methods*. Data encapsulation is one of the concepts of object-oriented programming (see Chapter 11). Data encapsulation functions as a security tool, because the data in the object is only available through the methods, which decreases the possibility of corrupting the data, and it reduces complexity (because the data is hidden within the module).

Consider the following example:

```
module student_class

implicit none

   private
   public :: create_student, get_mark

   type student_data
      character (len=50) :: name
      real :: mark
   end type student_data

  type (student_data), dimension (100) :: student

contains

   subroutine create_student (student_n, name, mark)

      ! here some code to set the name and mark of a student

   end subroutine create_student

   subroutine get_mark (name, mark)

      ! dummy arguments
      character (len=*), intent (in) :: name
      real, intent (out) :: mark

      ! local variables
      integer :: i

      do i = 1,100

         if (student(i)%name == name) then
            mark = student(i)%mark
         end if

      end do

   end subroutine get_mark

end module students
```

The student_class module defines a data type (student_data) to hold information of a student (name and a mark). Only the subroutines, create_student and get_mark, are

accessible from outside the module, all other module contents are private. Thus, one cannot obtain the mark of a student by writing:

```
mark1 = student(1)%mark
```

because the array student is private.

One has to use the public subroutine get_mark for this, as illustrated in the following program:

```
program student_list
   use student_class
   implicit none
   real :: mark
   call create_student (1, “Peter Peterson”, 8.5)
   call create_student (2,”John Johnson”, 6.3)
   call get_mark (“Peter Peterson”, mark)
   print*, “Peter Peterson:”, mark
end program student_list
```

Global data management – no more common blocks!

Procedures can communicate with each other via their argument lists. However, a program may consist of many procedures that require access to the same data. It would be convenient if this data were globally accessible to the whole program. In FORTRAN 77, this was accomplished by common blocks. However, modules can replace all uses of common blocks. Global data can be packed in a module, and all procedures requiring this data can simply use the module. Modules are much safer and cleaner than common blocks. Common blocks have no mechanisms to check errors, variables can be renamed implicitly, and there are no access restrictions. So, don’t use common blocks, use modules instead!

Summary

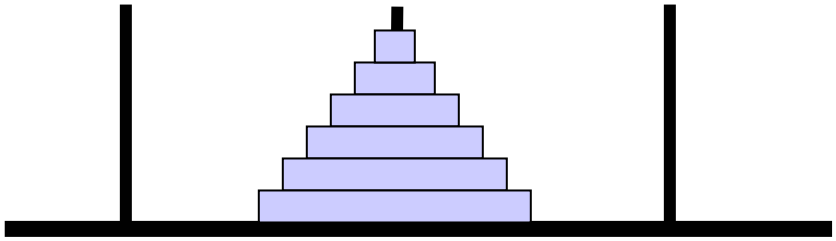
This chapter introduced a new and powerful feature in Fortran 90, modules. Modules are a means of packaging data and procedures. They make the old-fashioned common blocks obsolete. Modules provide a method to partition code into easily maintained packages, and allow some degree of object-oriented programming (we have seen an example of data hiding and encapsulation).

Exercises

Exercise 16, the Towers of Hanoi, is optional.

15. Finish the program student_list by adding to the student_class module the procedures create_student and delete_student.

16. The Towers of Hanoi.
    There are three poles. One of them contains discs having different widths stacked in ascending order; the other two poles are empty:



The goal is to move all the discs from the centre pole and stack them on one of the other poles in ascending order. You can only move the top disc of a stack, and you can only move one disc at a time. It is not allowed to stack a disc on top of one that is smaller.

You will have to figure out a way to represent the discs and their location, and an algorithm to move the discs. By printing the occupation of the towers after each move, you can check if your program works correctly.

Hint: The easiest algorithm uses a recursive function. By making the towers (their representation) global (by using modules), printing becomes a lot easier!

7. More on I/O

List-directed input/output

In Chapter 2 we have seen that we can transfer data from the keyboard into the program using read∗, and write characters to the screen via print∗. This simple form of I/O is called list-directed input/output. The ∗ in read∗ and print∗ means "free format", that is, the format is defined by the computer system, and is not under the control of the programmer. More control is given to the programmer via *formatted I/O*.

Formatted input/output

There are two forms of formatted input. The simplest one has the form:

```
read fmt, variable_list
```

Similarly, the simpler of the two forms of formatted output statements has the form:

```
print fmt, variable_list
```

Here, fmt denotes a *format specification*, and variable_list is a list of variables. This form of read and write reads from the keyboard and writes to the screen, respectively. The read* and print* we have used so far are a special case of this form.

Format specifications

A format specification defines in which format data is displayed. A format specification consists of a string, containing a list of *edit descriptors* in parentheses. An edit descriptor specifies the exact format (width, digits after decimal point) in which characters and numbers are displayed.

An example of format specification:

"(a10, i5, f5.3)"

- The a descriptor is for character variables. In aw, the number *w* specifies the field width. Thus, a10 means that 10 places will be used to display the character variable. If a field larger than the character variable is specified, the variable will be right-justified (*i.e.*, blanks will appear before the character variable). If no field width is specified, the width of the field is determined from the actual width of the character variable.
- The i descriptor is for integers. The number after the i is again the field width. Another form of the i descriptor is iw.m, where *w* defines the field width, and *m* specifies the minimum number of digits to be displayed, if necessary preceded by zeros.
- The f descriptor is for reals. In fw.d, *w* specifies the field width, and *d* specifies the number of digits after the decimal point. Thus, f5.3 will display the number 1.30065 as 1.301. Note that this number takes up 5 places: 1 digit before the decimal point, 3 digits after the decimal point, and the decimal point itself.
- Reals can also be displayed using the es descriptor, which displayed the real in scientific notation. The form is the same as for the f descriptor: esw.d. Thus, es10.3 would display the real 6.7345 as 6.734E+00 (with a leading blank, because of the field width 10).

As mentioned above, format specifiers are used in formatted I/O (in the following examples, *b* denotes a blank character):

```
pi = 3.1415927
print "(a, i3.3)", "Result = ", 1      ! gives: Result = 001
print "(f6.3)", pi                     ! gives: bbb3.142
print "(e16.4)", pi/100                ! gives: bbbbbb3.1416E-02
```

Repeat counts can be specified if more than 1 item is to be displayed with exactly the same format:

```
print "(3(f6.3)") x, y, z
```

More formatted I/O

As mentioned above, there are two different forms of the formatted input and output statements. The first form is the one we have just seen above:

read *fmt*, variable_list
print *fmt*, variable_list

The other form requires a unit number:

read (unit=*u*, fmt=*fmt*) variable_list
write (unit=*u*, fmt=*fmt*) variable_list

fmt is again a format specification, and *u* is a *unit number,* a number associated with a file (see next section).

(In Fortran 90, the "unit=" and "fmt=" in the read and write statements above are in principle not required – if they are omitted, the unit number has to be the first argument, and the format specification the second argument-, but they enhance readability. In F, the "unit=" and "fmt=" are required.)

Several optional specifiers can be specified as well, one of them is iostat, which can be used to recover from errors while reading or writing (it is very useful for checking errors while reading a file):

read (unit=*u*, fmt=*fmt,* iostat=*ios*) variable_list
write (unit=*u*, fmt=*fmt,* iostat=i*os*) variable_list

Here, ios must be an integer variable. If no errors occurred during reading or writing, the integer variable is set to 0, a positive integer means an error occurred, and a negative integer means an end-of-file condition occurred.

File handling

Most programs need to receive information, and need to output data. So far, we used the keyboard and screen to input and output information. However, other devices, such as a disk, tape, or cartridge, can be used as well. A collection of data on any of these devices is a *file*. To make a file available to the program (so that the progam can read data from, or write data to the file), it must be assigned a unit number. The open statement is used for this.

The open statement has the following form:

open (unit=*u*, file=*filename*, status=*st*, action=*act*)

- The unit number *u* is a positive integer (or integer expression). This specifier is required.
- The status *st* is a character expression. It can be "new" (the file should not yet exist, but will be created by the open statement), "old" (the file exists), "replace" (if the file doesn't exist, it is created, if the file already exists, it is deleted and a new file with the same name is created), or "scratch" (a file that is just used during execution of the program, and does not exist anymore afterwards).

- The *filename* is a character expression giving the name of the file. The file specifier is required if the status is "old", "new", or "replace", and it must *not* appear if the status is "scratch".
- The action *act* is a character expression as well; it can be "read" (file cannot be written into), "write" (cannot read from the file) or "readwrite" (no read and write restrictions).

These are the most common specifiers for the open statement (and are all required in F, but note that the filename must not be specified if the status is "scratch"). There are optional specifiers as well (access, iostat, form, recl, and position). The most useful of these is probably iostat, which should be set to an integer variable. This variable is set to zero if the open statement is correctly executed, and is set to a positive integer if an error occurred.

Unit numbers cannot be negative, and often the range 1-99 is allowed (although this is processor-specific). Generally, 5 is connected with console input, and 6 with output to screen. The unit number 0 is also often special. Thus, don't use 0, 5, and 6 for external files (files on disk).

We can read from and write to a disk file, if this file is opened and assigned a unit number:

```
write (unit=8, fmt = "(a2, 3(f10.6) )") atom_type, x, y, z
```

When a file is not longer needed, it should be closed. The close statement disconnects a file from a unit. The syntax is:

```
close (unit=u, iostat=ios, status=st)
```

The unit and iostat specifiers have the same meaning as above, status can be "keep" (the file will still exist after execution of this statement) or delete" (the file will be deleted). Only the unit specifier is required. If the status is not specified it is "keep", except for scratch files, for which it is "delete".

An example:

```
program file_example
   implicit none
   integer :: ierror
   open (unit=13, file="test.dat", status="new", action="write", iostat=ierror)
   if (ierror /= 0) then
      print*, "Failed to open test.dat"
      stop
   end if
   write (unit=13, fmt=*) "Hello world!"
   close (unit=13)
end programfile_example
```

This program creates a file called test.dat, opens it for writing, and writes a message into the file. After execution of the program the file will still exist. If the open failed, then execution of the program is stopped. It is good programming practice to test if the open statement was successful. If it had failed, the program would have crashed when it tried to write into it.

Internal files

A unit that is associated with an external device (like for example keyboard, screen, disk, or cartridge) is an *external file*. There are also *internal files*, and read and write can also read from and write into these. An internal file is a character variable. Contrary to external files, an internal file is not connected with a unit number.

Example:

```
character (len=8) :: word
character (len=2) :: ww
integer :: ierror

write (unit=word, fmt="(a)", iostat=ierror) "aabbccdd"
  ! the character variable "word" now contains the letters aabbccdd

read (unit=word, fmt="(a2)", iostat=ierror) ww
  ! the character variable "ww" now contains the letters aa
```

String manipulation functions

The following intrinsic functions to manipulate strings can be very useful:

trim
trim (string): returns the string with trailing blanks removed (the length of the string will be reduced)

adjustl
adjustl (string): removes leading blanks, and appends them to the end (so the length of the string remains constant).

adjustr
adjustr (string): removes trailing blanks, and inserts them at the front of the string.

index
index (string, substring): returns the starting position of a substring in a string, or zero if the substring does not occur in the string.

len
len (string) returns an integer with the value of the length of the string.

len_trim
len_trim (string): returns an integer with the value of the length of the string without trailing blanks.

Examples:

In the following examples, *b* denotes a blank character.

trim ("*bb*String*bb*") gives "*bb*String"
adjustl ("*bb*Tanja") gives "Tanja*bb*"
adjustr ("Word*bbbbb*") gives "*bbbbb*Word"
index ("Tanja van Mourik", "van") yields 7.
len_trim (Tanja*bbbb*) yields 5.

In the following example the program reads a file and uses the intrinsic index to search for the occurrence of the word "energy":

```
!! Example for reading an output file

program readout

   implicit none

   integer, parameter :: linelength = 120
   character (len=linelength) :: line
   integer :: ierror

   open (unit=10, file="test.dat", action ="read", status="old", iostat=ierror)
   if ( ierror /= 0 ) then
      print*, "Failed to open test.dat!"
      stop
   end if

   readfile : do

      read (unit=10, fmt="(a)", iostat=ierror) line

      if (ierror < 0 ) then
         print*, "end of file reached"
         exit readfile

      else if (ierror > 0) then
         print*, "Error during read"
         exit readfile

      else
         ! line has been read successfully
         ! check if it contains energy
            if (index (line, "energy") > 0) then
               print*, "energy found!"
               exit readfile
            end if
       end if

   end do readfile

   close (unit=12)

end program readout
```

The program reads the file "test.dat" line by line in the "endless" do loop readfile. Each time a line is read in, the program checks if the end of the file has occurred, or if an error accurred during reading. If either of these happened, then the do loop is exited.

Summary

In this chapter we learned formatted input/output and how to deal with files.

Exercises

17. Write a program with a function to eliminate blank characters from a string.

18. Gaussian is a well-known quantum chemical program package. The output of a calculation, for example a geometry optimisation using the MP2 (2^{nd}-order Møller-Plesset) method, contains a lot of data. Most of it is not very useful, and you may just be interested in the final optimised energy.

    Write a program that reads a Gaussian output file of an MP2 geometry optimisation, checks if the optimisation finished, and prints the optimised energy to another file. If you have access to Gaussian, create an output of a geometry optimisation (for example, H_2O using MP2 and the 6-31G* basis set), otherwise create a file that has the characteristics specified below, and test your program.

    After each geometry optimisation cycle, the energy is printed in a line like:

    E2 = -0.1199465917D+01 EUMP2 = -0.48968224768908D+03

    (The actual numbers of course depend on molecule, basis set, optimisation cycle). The MP2 energy is the one labelled EUMP2.

    When the geometry optimisation is finished, the program prints "Optimization completed". This happens after the last (optimised) energy is listed.

    (Alternatively, do the above using the output of your favourite computational chemistry program. If you do this, then provide an example of such an output with the solution to this exercise.)

8 Pointers

Pointers

The value of a particular data object, for example a real or an array, is stored somewhere in the computer's memory. Computer memory is divided into numbered memory locations. Each variable is located at a unique memory location, known as its address. Some objects require more storage space than others, so the address points to the starting location of the object. There is a clear distinction between the object's value and its location in memory. An object like an array may need a lot of memory storage space, but its address only requires a very small amount of memory.

In certain languages, like C and C++, a pointer simply holds the memory address of an object. A pointer in Fortran (which is a data object with the pointer attribute) is a bit more complicated. It contains more information about a particular object, such as its type, rank, extents, and memory address.

A pointer variable is declared with the pointer attribute. A pointer variable that is an array must be a *deferred-shape array*. In a deferred-shape array, only the rank (the number of dimensions) is specified. The bounds are specified by just a colon. The extent of the array in each dimension (*i.e.*, number of elements along a dimension) is determined when the pointer is allocated or assigned – see below.

```
integer, pointer :: p                  ! pointer to integer
real, pointer, dimension (:) :: rp     ! pointer to 1-dim real array
real, pointer, dimension (:,:) :: rp2  ! pointer to 2-dim real array
```

In contrast to a normal data object, a pointer has initially no space set aside for its contents. It can only be used after space has been associated with it. A target is the space that becomes associated with the pointer.

A pointer can point to:

- an area of dynamically allocated memory, as illustrated in the next section.
- a data object of the same type as the pointer, with the target attribute (see section on targets below)

Allocating space for a pointer

Space for a pointer object can be created by the allocate statement. This is the same statement we used before to allocate space for dynamic arrays (see Chapter 5).

```
program pointer1
   implicit none
   integer, pointer :: p1
   allocate (p1)
   p1 = 1
end program pointer1
```

The statement

```
integer, pointer :: p1
```

declares the pointer p1, but at this point it is not associated with a target. The allocate statement reserves space in memory. This space is the target that the pointer is now associated with.

After the statement

```
p1 = 1
```

the value of the target is 1.

The allocated storage space can be deallocated by the deallocate statement. It is a good idea to deallocate storage space that is not any more needed, to avoid accumulation of unused and unusable memory space.

Targets

A target object is an ordinary variable, with space set aside for it. However, to act as a target for a pointer is must be declared with the target attribute. This is to allow code

optimisation by the compiler. It is useful for the compiler to know that a variable that is not a pointer or a target cannot be associated to a pointer. Only an object with the target attribute can become the target of a pointer.

The program in the previous section can be rewritten as follows:

```
program pointer2
   implicit none
   integer, pointer :: p1
   integer, target :: t1
   p1 => t1
   p1 = 1
end program pointer2
```

After the statement

```
p1 => t1
```

p1 acts as an alias of t1. Changing p1 has the effect of changing t1 as well.

Association

The *association status* of a pointer can be *undefined*, *associated* and *disassociated*. If the associaton status is not undefined, it can be tested by the function associated. The function has 2 forms:

associated (ptr) returns the value true if the pointer ptr is associated with a target, and false otherwise.

associated (ptr, trgt) returns true of the pointer ptr is associated with the target trgt, and false otherwise.

So in the program in the previous section *before* the statement

```
p1 => t1
```

the association status is undefined, but *after* it both

associated (p1) and associated (p1, t1) would return true.

A pointer can be explicitly disassociated from a target by the nullify statement:

```
nullify (ptr)
```

It is a good idea to nullify pointers instead of leaving their status undefined, because they can then be tested with the associated function.

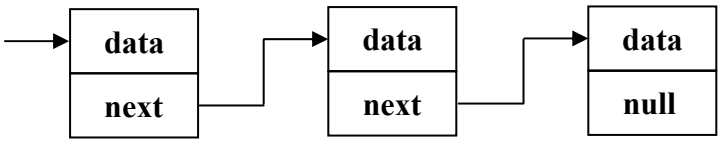
Nullify does not deallocate the targets (because there can be more than one pointer pointing to the same target). Deallocate implies nullification as well.

Linked lists

A *linked list* is a special kind of data storage structure. It consists of objects of derived type that are linked together by pointers. There are several kinds of linked lists (single-

linked lists, double-linked lists, binary trees). Here we will discuss the simplest, and most common of those, the single-linked list (usually referred to simply as a linked list).

Each element (also called node or link) of a linked list is an object of derived type that consists of a part with data and a pointer to the next object of the same list:



The pointer is of the same type as the other elements of the list. The derived type can for example be something like:

```
type node
   integer :: i
   real :: value
   type (node), pointer :: next
end type node
```

Linked lists are not unlike arrays, but there are differences. Linked lists can be allocated dynamically, so you don't need to know before the program is executed how many elements are needed (this also saves memory space). The size of the list can change during execution (links can be added and removed), and links can be added at any position in the list. The links are not necessarily stored contiguously in memory.

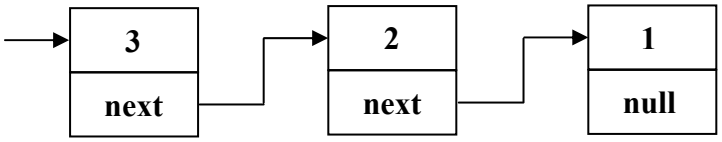
The "next" pointer of the last link in the list should be nullified. You also need a pointer (often referred to as *head pointer*) that refers to the first item in the list.

Consider the following example:

```
program linkedlist

   implicit none

   type :: link
      integer :: i
      type (link), pointer :: next
   end type link

   type (link), pointer :: first, current
   integer :: number

   nullify (first)
   nullify (current)

   ! read in a number, until 0 is entered
   do

      read*, number

      if (number == 0) then
         exit
      end if

      allocate (current)                    ! create new link
      current%i = number
      current%next => first                 ! point to previous link
      first => current                      ! update head pointer

   end do

   ! print the contents of the list
   current => first                         ! point to beginning of the list

   do

      if (.not. associated (current)) then  ! end of list reached
         exit
      end if

      print*, current%i
      current => current%next               ! go the next link in the list

   end do

end program linkedlist
```

In this program a link is defined that can hold an integer. The pointer "first" is the head pointer. In the first do loop, numbers are read in until a 0 is entered. After each number is read in, a new link is created and added before the previous link.

Thus, if the numbers 1, 2, and 3 are entered (in this order) the list will look like:



The contents of the list are printed in the second do loop. We start at the beginning of the list (current => first), and go from one link to the next (current => current%next), until the end of the list is reached (indicated by a not associated next pointer).

Exercises

19. Create a linked list. Each link contains a real number, which is read from screen in a do loop. After a number is read in, a new link must be created and added to the list in such a way that the list is sorted (*i.e.*, with increasing (or decreasing) values for the numbers). Preferably, adding the new link is done in a subroutine. Make also a subroutine to print the list, so you can check your program. To add the new link at the appropriate position in the list, you need to distinguish between the following cases:
    - First link. (Can be found out by the association status of the head pointer). If it's the first link, create the new link and make the head pointer point to the new link. (Don't forget to nullify the next pointer.)
    - Adding to the beginning. If the new number is smaller than the number in the first link, the new link needs to be the first one.
    - Second link that should not be before the first one. If you are adding the second link can be found out by testing the association status of the next pointer of the first link. The next pointer of the first link should point to the new link.
    - Adding to the middle. To find out where the new link has to be added to the list, you have to go through the list (in a similar way as in the second do loop in the example above), and compare the new number with the ones in the existing links. You need to keep track of three links: the new link, the current link (which is the one that goes through the list as in the example above), and the previous link. You should test if the new link should be added before the current one, and if so, the previous link has to point to the new one (that's why you need the previous link as well), and the new link has to point to the current link.
    - Before or after the last link. If the end of the list is reached, then you know that the new link should be added either directly before, or after the last link.

9 Numeric Precision

Fortran 90 allows the programmer to specify the required precision of real variables. If we declare the real variable x:

```
real :: x
```

then the x is represented with the default precision for the processor used. This precision can vary from computer to computer, depending on, among other things, the word length of the processor. Thus, a real will be more accurately represented on a 64-bit than on a 32-bit processor. In FORTRAN 77, the precision of real variables could be increased by using double precision for reals, which use two words instead of one to represent the numbers. In Fortran 90, the types integer, real, complex and logical have a "default kind" and a number of other kinds. How many other kinds there are for a certain type depends on the particular processor. Each kind has its own *kind type parameter*, which is a integer of positive value. For example, if, for a certain processor, a kind value of 8 yields a precision equivalent to the old double precision type of FORTRAN 77, then the following statement

```
real (kind = 8) :: x1
```

is equivalent to the FORTRAN 77 statement

```
double precision x1
```

However, this is not very portable, because the required kind value may be different on another computer. Although many computers use kind values that indicate the number of bytes used for storage of the variable, you cannot rely on this.

Fortran 90 has two intrinsic functions to obtain the kind value for the required precision of integers and reals: selected_int_kind and selected_real_kind, respectively.

The selected_real_kind function returns an integer that is the kind type parameter value necessary for a given decimal precision *p* and decimal exponent range *r*. The decimal precision is the number of significant digits, and the decimal exponent range specifies the smallest and largest representable number. The range is thus from 10^{-r} to 10^{+r}.

As an example:

```
ikind = selected_real_kind (p = 7, r = 99)
```

The integer ikind now contains the kind value needed for a precision of 7 decimal places, and a range of at least 10^{-99} to 10^{+99}.

The function selected_real_kind can be used in a number of different forms:

```
! if both precision and range are specified, the “p =” and “r =” are not needed
! the following two statements are therefore identical
        ikind = selected_real_kind (p = 7, r = 99)
        ikind = selected_real_kind (7, 99)

! If only the range is specified, the “r = “ is needed
        ikind = selected_real_kind (r = 99)

! if only one argument is used, it is the precision
! the following two statements are therefore identical
        ikind = selected_real_kind (p = 7)
        ikind = selected_real_kind (7)
```

If you want to use the ikind value in a type declaration statement, it has to be a constant (*i.e.*, declared with the parameter attribute). The real variable x declared in the following statement is precise to 7 decimal places, and has a range of at least 10^{-99} to 10^{+99}.

```
integer, parameter :: ikind = selected_real_kind (7, 99)
real (kind = ikind) :: x
```

If the kind value for the required precision or range is not available, a negative integer is returned.

The selected_int_kind function returns the lowest kind value needed for integers with the specified range:

```
integer, parameter :: ikind = selected_int_kind (10)
integer (kind = ikind) :: big_number
```

The integer big_number can now represent numbers from 10^{-10} to 10^{+10}. As for selected_real_kind, if the kind value for the required range is not available, a negative integer is returned.

Exercises

20. Write a program that declares a real with a precision of at least 7 decimal places and an integer that can represent the number 1000000000000.

10 Scope and Lifetime of Variables

Local variables in subroutines

The *scope* of an entity is that part of the program in which it is valid. The scope can be as large as the whole program, or as small as (part of) a single statement. Variables defined in subroutines are valid only in their subroutine *i.e.*, their scope is the subroutine. These variables are called local variables, and cannot be used outside the subroutine. The *lifetime* of a variable defined in a subroutine is as long as this subroutine, or any routine called by it, is running.

In the following example the variable int1 in the main program is *not* the same as int1 in the subroutine sub1 (they occupy different memory locations), and thus, the print statement in the main program would print 0 (and not 1).

```
program scope
  implicit none
  integer :: int1
    int1 = 0
    call sub1
    print*, int1
end program scope
subroutine sub1
  implicit none
  integer :: int1
    int1 = 1
    print*, int1
end subroutine sub1
```

The variable int1 in the subroutine sub1 goes out of scope at the end of the subroutine *i.e.*, it then does not exist anymore. Consider the following example:

```
program scope
  implicit none
    call sub1 (.true.)
    call sub1 (.false.)
end program scope
subroutine sub1 (first)
  implicit none
  logical, intent (in) :: first
  integer :: int1
  if (first) then
    int1 = 0
  else
    int1 = int1 + 1
  end if
  print*, int1
end subroutine sub1
```

The first time sub1 is called (with first = .true.), the variable int1 is set to 0. At the end of sub1, int1, being a local variable, goes out of scope and may be destroyed. Thus, one cannot rely on int1 containing the number 0 the second time sub1 is called. So the above program is actually wrong, although it may work with some compilers. If the

variable in a subroutine needs to be kept between successive calls to the subroutine, it should be given the attribute save:

```
subroutine sub1 (first)
  implicit none
  logical, intent (in) :: first
  integer, save :: int1
    [    ]
end program sub1
```

Note that initialisation of a variable in the declaration or in a data statement implicitly gives it the save status. However, it is clearer to explicitly include the save attribute also in these cases:

```
integer, save :: int1 = 0
```

One should not give a variable the save attribute if it does not need it, as it may impede optimisation by the compiler and it also makes *e.g.*, parallelisation more difficult.

Variables in modules

The lifetime of a variable declared in a module is as long as any routine using this module is running. Consider the following example:

```
module mod1
   implicit none
   integer :: int1
end module mod1
subroutine sub1 (first)
  use mod1
  implicit none
  logical, intent (in) :: first
  if (first) then
    int1 = 0
  else
    int1 = int1 + 1
  end if
end subroutine sub1
program prog1
   implicit none
   call sub1 (.true.)
   call sub1 (.false.)
end subroutine prog1
```

The integer int1 does not need to be declared in sub1, because it is already declared in the module mod1 (and sub1 uses the module). However, at the end of the subroutine, int1 goes out of scope (because there is not a subprogram anymore that uses the module mod1), and one cannot rely on int1 containing the number 0 the second time sub1 is called. So the above program is actually wrong (although it may work with some compilers). To make the above program standard conforming, one would have to either use the module in the main program (in addition to using it in the subroutine), or declare int1 with the save attribute.

11 Debugging

Debuggers

You have probably been using write or print statements to figure out why a program gives wrong results. The larger the program gets, the more cumbersome it is to search for errors like this. Debuggers are a much more powerful tool for stepping through the code and examining values.

A debugger lets you see each instruction in your program and lets you examine the values of variables and other data objects during execution of the program. The debugger loads the source code, and you run your program from within the debugger.

Most operating systems come with one compiler or the other. Graphical programming environments like Visual Fortran come with an integrated debugger. Full screen debuggers generally show the source code in a separate window.

Most debuggers have the following capabilities:

- Setting breakpoints. A breakpoint tells the debugger at which line the program should stop. Breakpoints allow analysis of the status of variables just before and after a critical line of code. Execution can be resumed after the variables have been examined.
- Stepping through a part of the source code line by line.
- Setting watch points. The debugger can show the value of a variable while the program is running, or break when a particular variable is read or written to.

To produce the information needed for the debugger the program should be compiled with the appropriate compiler flag (generally –g).

gdb

The GNU debugger, generally comes with Linux. Xxgdb is a graphical user interface to gdb for X window systems.

Some useful commands in gdb and xxgdb:

break: set a breakpoint.
run: begin execution of the program.
cont: continue execution
next: execute the next source line only, without stepping into any function call.
step: execute the next source line, stepping into a function if there is a function call.

Look at the man pages for more information on gdb.

dbx

The run, cont, next and step commands are the same as in gdb. Breakpoints can be set with stop:

stop [var]	stops execution when the value of variable *var* changes
stop in [proc]	stops execution when the procedure *proc* is entered.
Stop at [line]	sets a breakpoint at the specified line.

Look at the man pages for more information on dbx.

Exercises

21. Find an appropriate debugger on your computer. Recompile one of your programs with the debugger option specified. Use the debugger to step through the program. Set a few breakpoints and examine the value of variables during execution of the program.

Object-Oriented Programming

To discuss the object-oriented (OO) paradigm is beyond the scope of this manual, particular because Fortran is *not* an OO language. (Two of the most well-known OO languages are C++ and Java). However, object-oriented thinking is gaining ground in the programming world, and also Fortran 90 does support (or simulate) some of the OO ideas.

An "object" in a program is supposed to resemble a real-world object (like for example an animal, or a molecule). It has characteristics that distinguish it from other objects, and it can "behave" like its real-world counterpart. In Fortran 90 objects can be modelled with modules. Modules can contain data to define the characteristics of the object, and procedures to manipulate this data.

The four concepts of object-oriented programming are Data Encapsulation, Data Abstraction, Inheritance and Polymorphism. To be an object-oriented language, a language must support all four object-oriented concepts.

Fortran is in principle a "structured" programming language, but Fortran 90 does support some of the ideas of object-oriented thinking. Fortran 90's modules support data abstraction (grouping together of data and actions that are related to a single entity), data encapsulation (the process of hiding data within an "object", and allowing access to this data only through special procedures or member functions), but it lacks inheritance (deriving subclasses from a more general data type). Polymorphism refers to the ability to process objects differently depending on their data type (by redefinition of "methods" for derived types) and to the ability to perform the same operation on objects of different types. The latter type of polymorphism can be simulated in Fortran 90 through *overloading*.

The new upcoming Fortran 2000 standard completely supports object-oriented programming (including inheritance). However, the standard is not expected to be released before 2004.

To learn more about the OO paradigm, see for example:

"Object-Oriented Programming: A New Way of Thinking"
Donald W. and Lori A. MacVittie
CBM Books 1996.