ITS Information Technology Service

Guide 138

Version 3.2

An introduction to programming in Fortran 90

This guide provides an introduction to computer programming in the Fortran 90 programming language. The elements of programming are introduced in the context of Fortran 90 and a series of examples and exercises is used to illustrate their use. The aim of the course is to provide sufficient knowledge of programming and Fortran 90 to write straightforward programs.

The course is designed for those with little or no previous programming experience, but you will need to be able to work in linux or Unix and use a linux or Unix text editor.

University of Durham

Document code: **Guide 138**
Title: **An Introduction to programming in Fortran 90**
Version: **3.2**
Date: **October 2004**
Produced by: **University of Durham Information Technology Service**
This document was based on a Fortran 77 course written in the Department of Physics, University of Durham.

Conventions:
In this document, the following conventions are used:

- A `bold typewriter font` is used to represent the actual characters you type at the keyboard.
- A *`slanted typewriter font`* is used for items such as filenames which you should replace with particular instances.
- A typewriter font is used for what you see on the screen.
- A **bold font** is used to indicate named keys on the keyboard, for example, **Esc** and **Enter**, represent the keys marked Esc and Enter, respectively.
- Where two keys are separated by a forward slash (as in **Ctrl/B**, for example), press and hold down the first key (**Ctrl**), tap the second (**B**), and then release the first key.
- A **bold font** is also used where a technical term or command name is used in the text.

Contents

An introduction to Fortran 90: course outline

1. Introduction

Fortran is one of many programming languages available. The name Fortran is short for FORmula TRANslation and this guide is based on Fortran 90, which is a version agreed in 1990. Fortran 95, a later standard, was a minor revision of Fortran 90. The latest standard, Fortran 2003, is in its final draft stages.

Fortran was developed for general scientific computing and is a very popular language for this purpose. In 1996 it was estimated that Fortran was employed for more than 90% of scientific computation (see Scientific Computing World, April 1996). Fortran is not, however, particularly suitable as a non-scientific general-purpose language or for use in equipment control, commerce, text management etc where more appropriate alternatives are available.

Fortran 90 is available on the ITS Linux service, the UNIX service and on the Networked PC service. In this course you will use it on the ITS Linux service. You will need to be familiar with basic Linux commands (e.g. those covered in **Guide 169: An introduction to Linux**) and be able to use a Linux text editor, such as **pico**, **emacs** or **vi**.

About the course

This course provides an introduction to the Fortran 90 programming language. It should provide you with enough knowledge to write straightforward Fortran programs and you should also gain some general experience which can usefully be applied when using any programming language. The course is constructed from five parts:

Part 1 Getting started: programming basics

Part 2 Input and output, and using intrinsic functions

Part 3 Arrays: vectors and matrices

Part 4 Program control: **do** loops and **if** statements

Part 5 Subprograms: functions and subroutines

If you receive these notes on an ITS course, they will be issued by part. Please bring the notes for previous parts of the course with you to future sessions so that you have them for reference.

If, at the end of the course, you wish to know more about Fortran 90, many books and on-line tutorials have been written on the subject. Some suggestions are given at the end of Part 5.

An introduction to Fortran 90: PART 1

2. Programming basics

This section describes the structure and contents of a Fortran 90 program. A program is simply a set of instructions that tell a computer to do a particular task. To create a program that works, and works efficiently, you will need to do the following before you start writing the program:

1) Make sure that you understand the aims of the program.

2) Decide on the information that the program will need as input and that it should produce as output.

3) Make sure that you understand how the computations will be done (i.e. that you understand the algorithms to be used) and the order in which they should be done.

It is very easy to skip one or more of steps 1 - 3 and start writing a program without really thinking about how it will work. Unless the program is very small, this will probably lead to a program that is badly structured and difficult to follow. Most programs do not work perfectly first time, but they are more likely to work and will be easier to correct if they are well structured from the beginning.

So, once you have completed steps 1 - 3,

4) Create a program that will solve the specified problem with the algorithm(s) and input/output requirements that you have identified.

5) Test the program thoroughly.

2.1 The main parts of a Fortran 90 program

A Fortran 90 program has four main elements, in this order:

Program name

The first line of the program gives the program's name, e.g.

```
program fred2
```

The program name allows you to recognise the program by looking at the version that you have typed into the computer. Later versions of the program might be called, for example,

```
program fred3
```

or

```
program fred4
```

In principle the program name is optional, but during this course you should always give a name to your programs.

Initialisation section: declaration of variables

The initialisation section sets some rules for the program but does not carry out calculations. In particular, it is used to *declare* all the variables that will be used to store numbers, etc, within your program. In Fortran 90 there are several pre-defined variable types such as **integer**, **real** and **character**, and it is even possible to define new types. The declaration statements set the types of your variables and can also be used to set initial values. The Fortran rules do not insist that you declare variables, but it is good programming and helps avoid many errors.

Main program body

The main program body contains all the executable Fortran statements that will carry out the task you wish to perform. Statements are generally executed in order, from top to bottom. The last statement must be an **end** statement. In many cases the efficiency and appearance of the main program can be aided by:

Subprogram(s)

The structure of a program can be made much clearer if blocks of instructions that perform particular tasks are moved out of the main program and into *subprograms*. Subprograms are also very useful if the same block of instructions is needed several times.

2.2 The layout of Fortran 90 statements

A Fortran 90 program consists of a series of executable and non-executable statements. *Executable* statements are ones that cause the computer to perform some desired operation, e.g. the addition of two numbers, whereas *non-executable* statements provide information which enables the proper operation of the program, e.g. the definition of variables.

Whether it is part of an executable or non-executable statement, each line in a Fortran 90 program must conform to certain rules about layout:

- A line can be up to 132 characters long (including spaces).
- Any line can start with leading spaces to improve layout.
- An **&** at the end of a line indicates that the statement continues on the next line. If the item to be continued is a character constant or a format statement (both discussed later in the course), the next line should start with a second **&**.
- If a line contains an exclamation mark (**!**), everything from the exclamation mark to the end of the line is a comment and will not be executed. Every comment line must begin with this character.
- Several statements can appear on a line, separated by semi-colons (**;**).

Note: earlier versions of Fortran used a much more rigid layout. This is still permitted in Fortran 90 but is no longer necessary. Details of the older layout style are given in section 15.1. The newer, 'free format' layout is used throughout this guide.

3. Data types

The data that are handled by a program fall into two main types. *Constants* have a fixed value, while *variables*, in which the program will store its input, output, constants and intermediate results, may change value during execution of the program.

3.1 Constants

Any constant must have a *type* so that the computer knows how to store and handle it. The range of numbers permitted in each type depends on the computer: the descriptions given below are common on machines with a 32-bit operating system. The main types of constant are:

Integer Integers are normally stored in 4 bytes (that is, 32 bits, i.e. 32 binary 0s and 1s) of storage space. This allows integers from -2147483647 (2^{31}) to +2147483647 (2^{31} - 1) to be represented.

Real Floating-point real numbers also have a default storage allocation of 4 bytes. The sign, mantissa and exponent must all be held in this space. Depending on the Fortran 90 implementation; real numbers can lie typically between approximately 10^{-37} – 10^{38}. Four-byte floating point numbers have *only about 7 significant digits*.

Double precision Double precision numbers are similar to real numbers but are allocated twice as much storage space, 8 bytes, so that they can hold more digits in the mantissa. They have about 15 significant figures and can lie in the approximate range 10^{-307} - 10^{308}.

Character Character variables can be used to hold standard text/ASCII characters with one byte per character and N bytes in total, where N is an integer. Unless N is defined in the declaration of the variable, the variable can hold only 1 character.

Logical Logical variables have a value of either **.true.** or **.false.** . They take storage space of 4 bytes.

Complex Two real (4 byte) numbers stored as a pair and treated as the real and imaginary parts of a complex number.

The following examples show you how to enter constants correctly.

3.1.1 Integers

Any number without a decimal point falling between the prescribed limits is a valid integer, e.g:

```
-3478
0
567890
+12345678
```

The following are not valid integers:

`-1,000`	(commas not allowed)
`987.`	(contains a decimal point)
`987654321098`	(too big)

3.1.2 Reals

Real numbers contain a decimal point and lie within the prescribed range such as:

`0.0123`	(can also be written as `1.23E-02`)
`0.0`	(can also be written as `0.0E0`)
`-23456.0`	(can also be written as `-2.3456E4`)
`+987652.0`	(can also be written as `9.87652E+05`)

Examples of *illegal* **real** constants are:

`-10`	(no decimal point, integer)
`1,123.`	(commas not allowed)
`145678E4`	(no decimal point in mantissa)
`666888.E8.2`	(decimal points not allowed in exponent)

3.1.3 Double Precision

These follow the same basic rules as for **real** numbers but **D** must be used instead of **E** to indicate the exponent. For example, `1.23D-02` and `0.0123D0` represent the double precision version of `0.0123`.

3.1.4 Character

Character constants must be contained within single quotes (apostrophes), for example:

```
'This is a 31 character constant'
'       '
'45.68'
```

The following are not valid character constants:

```
Invalid character constant    (no quotes)
'Another one                  (unpaired quote)
```

To include an apostrophe within the character constant, two apostrophes should be used e.g.

```
'Fortran 90 solved all of Julie''s problems'
```

The `''` pair will be interpreted as a single apostrophe within the character string.

3.1.5 Logical

The only valid **logical** constants are **.true.** and **.false. (or .TRUE. and .FALSE.)**.

3.1.6 Complex

Complex constants are written as a bracketed pair of valid real numbers separated by a comma, e.g.:

```
(1.234,-6.5E-3)
```

where, in this example, `1.234` is the real part of the complex constant and `-0.0065` is the imaginary component.

3.2 Variables

Variables are where your program will store its input, output, constants and intermediate results. In standard Fortran 90, variable names may contain alphabetic and numeric characters and underscores but must begin with an alphabetic character and be no longer than 31 characters long in total. So **loop3** and **MyAnswer** are valid variable names but **123x** and **_abc** are not. In general it is helpful to give variables sensible names that suggest their purpose.

You should be aware that unlike some other programming languages Fortran does *not* distinguish between upper and lower case characters, so a variable called **NUMBER** is entirely equivalent to one called **number** or **NUMber** etc. Early versions of Fortran used upper case only, but this is no longer necessary and you need not follow this convention during the course.

Like constants, variables have to have a *type*. Any valid constant value can be assigned to a Fortran variable, provided that the type of the constant is compatible with the variable's type.

At the beginning of a program you should *declare* each variable and its type. The declaration is simply a statement of the variable's type and some optional extra details such as an initial value.

If you miss out the declaration of a variable, Fortran 90 will give it a type according to its name: undeclared variables with names beginning with the letters I,J,K,L,M,N will be given type **integer** whereas all others (A to H and O to Z) will be type **real.** This 'implicit typing' rule is convenient but also causes problems, since it means that Fortran will accept any mis-spelled or undeclared variables instead of warning you. This is discussed further in section 12. To instruct Fortran to reject any variables that have not been declared, include the line:

```
implicit none
```

before any other declarations. The **implicit none** statement should be used in all of the programs in this course.

As well as using **implicit none**, it is strongly advised that you comply with the naming conventions of implicit typing when naming variables**.** This can be useful if (when!) you need to resolve problems in your programs.

4. How to write, process and run a program

There are four stages to creating a Fortran 90 program:

1) Create and save the program using a text editor. The file you create is called the *source code* and should have a name that ends in **.f90**, e.g. **fred.f90**.

2) *Compile* the code into an intermediate format, called an *object* file. Compilation is done by the command **pgf90**. During compilation your program is checked for syntax errors and, if none are found, a file is created with a name ending in **.o**, e.g. **fred.o**.

3) *Link* the file into a final, executable form of your program. If compilation was successful, the **pgf90** command will next link your program with any *libraries* that it needs. If linking is successful, the object file will be removed and an executable file will be created. The default name for the file will be **a.out**, but you can also specify a name, e.g. **fred**. Whereas the original source code that you typed in should normally be understandable to any Fortran 90 compiler, the final, executable form of your program is specific to a particular machine type.

4) Run the program.

4.1 Writing the program

In the following sections of this course, you will create a number of small programs. To keep the files separate from the rest of your work, you might want to create a directory now in which to store the files. Then **cd** to the new directory and work from there:

```
cd
mkdir Fortran
cd Fortran
```

Use a text editor (e.g. pico or emacs) to type in the following program and save it as a file called **mult1.f90**.

```
program mult1
implicit none
integer:: i,j,k
!
! This simple Fortran program multiplies two integers.
! It then displays the integers and their product.
!
i = 5
j = 8
k = i * j
write(*,*)i,j,k
stop
end program mult1
```

Note the following:

Line 1 The file and program names do not have to be the same. However, it is sensible to choose meaningful names for both so that you know which program is in which file and have an indication of the purpose of the program.

Line 2 The *implicit none* line tells the compiler not to use the implicit rules for variable types/names. Its effect is that any undeclared or mis-spelled variables will cause an error when the program is compiled.

Line 3 This *declaration statement* reserves storage space for three integer variables called *i*, *j* and *k*.

Lines 4-7 *Comment statements* have been used to document the program.

Lines 8-9 The *assignment statement* **i = 5** places the value 5 into the variable *i* and similarly the statement **j = 8** gives *j* the value 8.

Line 10 The *assignment statement* **k = i * j** multiplies the values of variables *i* and *j* on the right and assigns the result to the variable *k* on the left. Notice that the result variable must be on the left.

Line 11 A **write** *statement* is used to display the contents of the three variables *i, j* and *k*. Note that commas must be used to separate the items in the list of variables. The (*,*) part contains information about the *destination* for the output and the *format* of the output. The * symbols mean that we accept the defaults: the default destination is the screen and we will accept the default format.

Line 12 When a running program arrives at the **stop** statement, execution is terminated and the word STOP is displayed on the screen. The STOP statement is not compulsory. Although it is not recommended, a program may have several STOP

statements numbered STOP 1, STOP 2, etc, at different places within the program.

Line 13 The **end** statement indicates the end of the program or subprogram. It is good practice to include the name of the program in this line, as shown.

When you have typed in the program, compile, link and run the program as described in the next section.

4.2 Compilation and linking

Once you have created your source file **mult1.f90**, you can compile and link it in one step using the **pgf90** command. At a Linux command prompt, type:

```
pgf90 -o mult1 mult1.f90
```

where the basic command is **pgf90 mult1.f90** and the option **-o mult1** is used to name the output file **mult1**. If you omit this option, your executable file will be called **a.out**, regardless of the name of your source file.

Note: on the ITS Sun UNIX service, the compilation and linking command is **f90**.

If the compilation and linking process was successful, **pgf90** will run without displaying any messages. If any errors occurred during compilation or linking, error messages will be displayed. Go back to the text editor and correct the errors in **mult1.f90**, then save, re-compile and re-link.

Once the compilation and linking have been completed successfully, you will have an executable file called **mult1**.

4.3 Running the program

To run the mult1 executable, simply type its name:

```
mult1
```

If your program compiled, linked and ran first time without errors then go back to the editor and deliberately introduce errors in order to see what happens - e.g. try changing **integer** to **intger** in the declaration statement and try changing **i** to **ii** in line 10. What happens if you leave this second mistake and remove the **implicit none** statement? Make sure that you remember to correct your program afterwards!

4.4 Removing old files

At the end of this first exercise you will have generated some files. The **mult1.f90** source file that you typed is useful to keep for future reference, but you can delete the executable file **mult1**, which is using up space. If you have a **mult1.o** file, that can be removed too. The executable and object files can be regenerated from **mult1.f90** if you need them again.

In the next exercise you will need to modify your mult1 program. Instead of editing **mult1.f90**, it is a good idea to copy **mult1.f90** to another file, e.g. **mult2.f90**, then work on this new file. When you are developing a new program in a series of stages it is always a good idea to take copies as you go along and leave previous (working) versions in case your modified program does not work and you cannot remember exactly how the original was written! So **always keep backup copies of your source files.**

Exercise

In this exercise you will write a simple program that reads any two integers typed on the keyboard and writes the two integers and their product on the screen. To work through this section, first make a copy of **mult1.f90** and call the copy **mult2.f90**.

One of the problems with **mult1** is that if you want to multiply any other pair of integers, instead of 5 and 8, then you need to modify the program each time. An alternative, more efficient approach is to modify the program so that you can input any two integers and then multiply these. In order to do this you should remove lines 8 and 9 from **mult2.f90**, i.e. the lines:

```
i = 5
j = 8
```

and replace them with the single line:

```
read(*,*)i,j
```

This **read** statement tells the computer to expect to receive two values from the keyboard when the program runs. When it receives the two values, it will place them in order in the integer variables **i** and **j**. Note that the list of variables in a **read** statement must be separated by commas.

In this example the first * in the **brackets** is an instruction to use the default source of information, which is the keyboard, and the second * tells the computer to interpret the two values in an appropriate default manner - in this case as integers because *i* and *j* have been declared as integers. Later in the course you will see how to specify a different source of information (e.g. a file) and a particular format for the incoming information.

When you have made the necessary changes to **mult2.f90**, compile, link and run your new program. The program will pause to allow you to input the information required. Type in **5,8** or **5 8** and then press the **Enter** key and you should then obtain the same result as with **mult1**. With this new program, however, you can multiply *any* two integers.

This program is still not very "user-friendly". For example there is no indication that the program is waiting for the input of numbers. To improve this, include the following **write** statement before the **read** statement:

```
write(*,*) 'Enter the two integers to be multiplied'
```

Anything that is written between the two apostrophes will appear on the screen when the modified program is run. Try it!

More user-friendly output can also be obtained by mixing text and results in the following manner. Try replacing the simple output statement:

```
write(*,*)i,j,k
```

with the following version:

```
write(*,*)' The product of ',i,' and ',j,' is ',k
```

As you can see, anything between pairs of apostrophes is written literally to the screen as a character string, while the other items (*i*, *j* and *k*) which are not between apostrophes are recognised as variables and their *value* is written.

Later in the course you will see how to customise the format of your output using a **format** statement.

Exercise

Modify **mult2.f90** so that it operates with **real** variables *x,y,z* instead of **integer** *i,j,k* and name the altered program **mult3.f90.** How does the answer differ from the answer given by **mult2**?

Next, modify **mult3.f90** so that it operates with **double precision** variables *x,y,z* instead of **real** *x,y,z* and name the altered program **mult4.f90**. What changes do you notice in your answers?

5. Converting between types of variable

In the exercises above you experimented with some different types of variable. It is important to use appropriate variable types in your program, or it may not work properly. For example, the following program shows what happens when two integers are divided. Type the program into a file named **divide1.f90**:

```
program divide1
implicit none
integer:: i,j
real:: x
!
! Program to demonstrate integer division.
!
write(*,*)' Enter two integers'
read(*,*)i,j
x = i / j
write(*,*) i,' divided by ',j,' is ',x
stop
end program divide1
```

Compile and link this program, and then run it for some trial values of *i* and *j*. You will see that the results are only correct when *i* is an exact

multiple of *j*. In any other case, the result is truncated to an integer even though *x* is real. This happens because of the way the statement is calculated. The right hand side of the line *x = i/j* is calculated first. Both *i* and *j* are integers, so the result of *i/j* is also an integer. The integer result is then converted to a real number to be stored in *x*, but the truncation has already occurred. Accidental integer division is a common error in programming.

In order to obtain the true (possibly non-integer) ratio of any pair of integers, copy your program to a new file **divide2.f90** and alter the line

```
x = i / j
```

of your new program to:

```
x=real(i)/real(j)
```

real converts an integer value into a real value and thus the division now involves two real numbers. Check that your new program gives the correct answer. Note that it would not have been sufficient to write **real(i/j)**. Why not?

real is called an **intrinsic function** because it is a function that is built-in to Fortran. Some other intrinsic functions for conversion between different number types are:

`dble`	transform a variable to double precision
`int`	truncate a real number to an integer
`nint`	round a real number to the nearest integer

Reminder! At this point you will have a number of files taking up disk space. Delete any that you do not need (but keep the **.f90** source files for later reference).

Exercise

Write a program (**inv.f90**) to read a real number, take the inverse and show the result on the screen. Write a second version (**inv2.f90**) that works in double precision**.** Run the two versions of your program, using the number 7 as input, and compare the results.

6. The hierarchy of operations in Fortran

You have already seen that the order of operations matters. In general, the computer takes the following steps when executing an assignment statement:

1) it calculates the result on the right hand side of the statement in a form appropriate to the types involved,

2) it looks at the type of the variable on the left and finally,

3) it converts the result on the right in order to assign a value to the variable on the left hand side.

The first of these steps often involves evaluating an arithmetic expression. When a general Fortran arithmetic expression is computed there is a strict order in which the component parts of the expression are evaluated. The order is:

1) Terms in (round) **brackets** starting from the innermost brackets and working outwards.

2) **Exponentials** working from right to left (Note that x^y is written **x**y** in Fortran).

3) **Multiplication** (denoted by a single *) and division, working from left to right.

4) **Additions** and **subtractions** working from left to right.

Note that consecutive mathematical operators are not allowed. Some examples of *invalid* expressions and corrected versions are:

Incorrect	*Problem*	*Correct*
`A**-B`	consecutive operators	`A**(-B)`
`A(B+3.6)`	missing operator	`A*(B+3.6)`
`A*2.75-B*(C+D))`	unpaired brackets	`A*(2.75-B*(C+D))`

Exercise

Write a program, **order.f90**, to evaluate the following three expressions:

```
x=a*b+c*d+e/f**g
y=a*(b+c)*d+(e/f)**g
z=a*(b+c)*(d+e)/f**g
```

where all variables are of type **real**. Set trial values for the variables and convince yourself that the expressions are evaluated according to the described hierarchy.

An introduction to Fortran 90: PART 2

7. About input and output

So far in the course you have used simple "free format" **read** and **write** statements to control input from the keyboard and output to the screen. This section shows how to extend these to:

- Accept input from sources other than the keyboard and direct output to locations other than the screen (e.g. to a file).
- Specify the precise format of the input or output.

7.1 Redirection of input/output

By default, input is read from the keyboard and output is displayed on the screen. These defaults are indicated by the first * in the **read(*,*)** or **write(*,*)** statement. When you want to read from somewhere else, or to write to somewhere else, this * should be replaced with a *unit identifier* which identifies a particular location for the input or output. The unit identifier is simply an integer or an integer expression. It is best to use small, positive integers (less than 64) because the permissible range varies between systems.

Note that, by tradition, unit **5** is used as a default for the standard input unit (almost always the keyboard), **6** is a default for the standard output unit (almost always the screen) and **0** is the default unit for errors, so when you use other files/devices it is normal not to use these numbers.

Apart from the keyboard for input and the screen for output, the most frequent location for input/output is a file. Before a file can be accessed from a **read** or **write** statement in your program, the program must make a connection between the actual file concerned and the Fortran unit. This is done with an **open** statement.

Example

Imagine that you wish to read some input from a file called **info.dat** which is in the directory **/scratch/share/dxy3abc**. You could use 10 as the number for this input unit. In order to make a connection to the file, you would include the following statement in the program, *before* the first time you try to **read** from the file:

```
open(10,file='/scratch/share/dxy3abc/info.dat')
```

Notice that the filename **/scratch/share/dxy3abc/info.dat** is in quotes as is normally the case for character strings in Fortran. You can also use the lengthier, more explicit version of the statement:

```
open(unit=10,file='/scratch/share/dxy3abc/info.dat')
```

After the **open** statement, any subsequent references to unit 10 in **read** statements will automatically be directed to this file. So, for example, in order to read data from **info.dat** you might use:

```
read(10,*)k,l,m
```

to read free-format data from the file into the three integer variables **k, l** and **m**. Each successive **read** will (normally) cause input to be read from the next line in the file.

Writing to a specific location is done in exactly the same way as reading. In order to write to unit 11 (a file), we would first need to **open** the file and then use, for example:

```
write(11,*)k,l,m
```

As with **read** statements, each successive **write** statement will normally direct output to the next line in the file.

When a file is no longer required (e.g. when the program has finished reading data from it) the connection between the unit identifier and the file should be closed by making use of the **close** statement. To close the connection with unit 10, the statement would be:

```
close(10)
```

Note: output files may appear incomplete or empty until the **close** statement has been executed or the program exits.

Exercise

As a demonstration of output to a file, type the following program into a file called **output1.f90**, then compile, link and run the program and inspect the results in the file **out.dat**, which will be in your current directory. Note that the program writes no output on the screen, only to the file.

```
program output1
implicit none
integer:: iyears, imonths, iage
character (LEN=30):: name
!
! Simple program to send output to a file.
!
write(*,*)' Hello. What is your name (type it in single quotes)'
read(*,*)name
write(*,*)' How old are you (years and months)?'
read(*,*)iyears, imonths
iage = iyears * 12 + imonths
open(1,file='out.dat')
write(1,*)name, ' is ',iage,' months old!'
close(1)
stop
end program output
```

7.2 Formatting input and output

Sometimes the default format that Fortran uses for input and output is not sufficient. For example, you may need to read data from a file that contains a specific layout, or you may want a neater layout for your output. You can control the format of input or output with a *format statement*. Format statements have a particular syntax, for example:

```
10 format(i2,i4)
```

Every format statement begins with a number that identifies it, called the *statement label*. After the statement label there is a space and then the word **format**. The expression in the brackets defines the precise format in which the data are to be input/output. The example above uses I-format, which is for integers. The *i2* indicates that the first item is an integer composed of two digits, and the *i4* indicates that the next item is a four-digit integer. Unlike executable statements, a non-executable **format** statement like this one can appear anywhere within the program, before or after the associated **read**/**write** statement. It can also be used with more than one **read**/**write**. In a program with many **format** statements, it is a good idea to group them together at the beginning or end of the program so that it is easy to find them.

Note: if a format statement continues over more than one line, it should have a **&** character at the end of the first line *and* another **&** at the beginning of the next line.

To make use of a format statement, replace the second * in a **read** statement or a **write** statement by the label for the **format** statement. For example:

```
write(*,10) i,j,k

10 format(i2,i4,i10)
```

This example again uses I-format. It writes the variables **i**, **j** and **k** as a 2-digit integer, a 4-digit integer and a 10-digit integer.

When you use a **format** statement with a **write** statement, the formats do not necessarily have to match the number of digits in each variable. If a number is smaller than the size allowed in the format statement, it will be padded with spaces (to the left of the number). If it is too large for the format, then it will be replaced by a row of ***** -- if this happens you will probably need to edit your program to solve the problem.

When you use a **format** statement with a **read** statement, be more careful. If the format you specify is too small for the number you want to read, it will be truncated (i.e. wrong). If the format is too large, you may read characters that should not be included.

Note: for short **format** statements that you will use only once, the format can be inserted directly into the **read**/**write** statement in the following way:

```
write(*,'(i2,i4, i10)') i,j,k
```

The I-format that you have seen so far is for integers. Other format types are used for the other variable types. The most commonly encountered formats are:

Type	Syntax	Example	Data/Output	Description

integer	`Iw`	`i3`	`123`
			` 12`
			`-98`
				w is the total number of characters, i.e. the *width.*

real	`Fw.d`	`f8.3`	`1234.678` `-234.678`	*w* is the total number of characters *including the decimal point if present* and *d* is the number of digits after the decimal point
character	`Aw`	`a5`	Abcde	*w* is the total number of characters
Space	`x`	`1x`		spaces (when writing), or characters to skip (reading)

In the descriptions of the formats, the term **character** is used to indicate the symbols on the screen, whether they are digits, letters or spaces. In the case of numbers the characters are interpreted in the form of the appropriate **integer** or **real** numbers. *Whenever the decimal point is included, it counts as one character in the field width.* The total field width includes any minus sign and any leading spaces.

To specify multiples of a format, prefix the format specification with the number involved. For example, to read one 2-digit integer followed by two 8-digit integers, you could use the format statement:

```
10 format(i2,2i8)
```

It is also possible to specify where new lines start, using the **/** symbol. To read four 3-digit integers from line 1 of the file and another two from line 2, the **format** statement could be:

```
10 format(4i3/2i3)
```

Any data on line 1 beyond the first four integers will be ignored.

Note that a comma is required to separate the items in a **format** statement, *except* when there is a **/** between them to indicate that data continue on the next line.

Normally the number of variables in a **read** or **write** statement should match the number of items in the **format** statement. If there are more variables than items in the **format** statement, as in the example below, the format is simply repeated until all the variables have been processed.

```
read(1,10) i,j,k,l,m,n
10 format(3i2)
```

Here, *i*, *j* and *k* will be read from one line and *l*, *m* and *n* would be read from the next line in the same format.

Note that on some computers, unless the output is being directed to a file, the first character of each line may be treated as a "carriage control" character when the output is displayed or printed and, unless you ensure that the first character is a space, the output may not look as you intended. (In particular, you may lose a minus sign or the first character of the field!). For this reason, you will find that programs often begin output formats with a space (1x). For more information on this topic, please consult the sources listed at the end of the course.

Exercise

When you *read* data, you will often find it easier not to use a format statement, because the data are forced into the format that you specify. To illustrate this point, type the following program into a file called **format1.f90**

```
program format1
implicit none
integer::  i, j
real::  x
character(len=10):: mychars
!
! Simple program to demonstrate the use of the format statement
!
write(*,*)' Enter digits 0 to 9 twice in succession '
read(*,100)mychars, i, x, j
100  format(1x, a5, i5, f6.2, i3)
write(*,*)mychars, i, j, x
stop
end program format1
```

Run the program and enter **01234567890123456789** when requested.

The *1x* in the **format** statement causes the first digit (0) to be skipped. The next 5 digits, 1 to 5, are interpreted as characters (a5) and are entered into the first 5 places of the 10-character variable, *mychars*. The integer value **67890** is placed in variable *i* according to the *i5* specification. The next 6 digits, **123456**, are interpreted as **1234.56** because of the *f6.2* and are placed in *x* and, finally, *j* is given the value **789** as required by the *i3* format. Check this by looking at your output. Then see what happens if

(a) you change the **format** statement to

```
100 format(a8, i4, 2x, f4.2, i2)
```

(b) you change the **format** statement so that it does not match the variable types, e.g:

```
100 format(a8, f4.2, 2x, i4, i2)
```

In the exercise above, you used format *f6.2* to deal with the **real** variable *x*. This defined the decimal point to be between the 4th and 5th digit in the format and it was inserted there even though there was no decimal point in the input data. In practice, the decimal point should usually be included in the input data: provided that the number fits into the field width, it will be

recognised correctly. If the number is negative then the minus sign must also fit within the field.

7.3 *E* format and *D* format

A frequent problem with formatted output is that a number is too big to fit into the specified field width. For example, **9999999.0** does not fit into format F6.2, which can only deal with the correct output of numbers up to **999.99**. When this happens, the entire number is replaced in the output by asterisks, e.g. ******. If you are not sure how big an answer will be, consider whether it would be better to display it with E-format (exponential format), which is the best way to display numbers that are (or could be) very large or very small.

For example, a number **1.234567×10^{-12}** displayed in E14.7 format would appear as:

```
0.1234567E-11
```

where the E14.7 specifies 7 characters after the decimal point and 7 other characters. There is an equivalent D-descriptor form for double precision numbers. For example, D14.7 format would give output such as:

```
0.1234567D-11
```

Note that the total width of E- and D- format fields (14 in these examples) must be big enough to accommodate any minus sign, the decimal point and the E-*nn*, so it is recommended that the total field width should always be **at least 7 places more** than the number of digits required after the decimal point.

Exercise

To practise using format statements, make a copy of your program **output1.f90** and call the copy **output2.f90**. In output2.f90, change the line:

```
write(1,*) name, ' is ',iage,' months old!'
```

to

```
write(1,100) name, iage
100 format(a30,' is ',i4,' months old!')
```

Check that the output matches the format statement.

8. More intrinsic functions

You have already seen the intrinsic functions **real**, **dble**, **int** and **nint**, which converted variable values from one type to another. Fortran 90 has over 100 intrinsic functions. Some of the most frequently used ones are:

Function	Action of function
`sqrt(x)`	square root of x
`abs(x)`	absolute value of x
`sin(x)`	sine of x (x in radians)
`cos(x)`	cosine of x (x in radians)
`tan(x)`	tangent of x (x in radians)
`asin(x)`	$\sin^{-1}(x)$ (result $0 \rightarrow \pi$)
`acos(x)`	$\cos^{-1}(x)$ (result $-\pi/2 \rightarrow \pi/2$)
`atan(x)`	$\tan^{-1}(x)$ (result $-\pi/2 \rightarrow \pi/2$)
`exp(x)`	e^x
`alog(x)`	$\log_e(x)$
`alog10(x)`	$\log_{10}(x)$

In this list, *x* represents any real or double precision variable, constant or expression. These functions will not accept integer arguments. The output of the function will be real for real *x* and double precision for double precision *x*.

Note: all trigonometric functions use radians and not degrees. 2π radians = 360 degrees.

Exercises

1) The formula for calculating compound interest is:

$$final = start\left(1+\frac{rate}{100}\right)^{nyears}$$

where *final* is the final value, *start* is the start value, *rate* is the annual interest rate in percent, and *nyears* is the number of years. Write a program called **money.f90** that calculates the value of a £1000 investment after 5 years, for interest rates of 2, 4, 6 and 8% and writes the interest rates and results neatly on the screen and in a file called **mymoney**.

2) Write a program readmoney**.f90** which reads the interest rates and final sums from the file **mymoney** and then displays them on the screen.

3) Write a program called **trig.f90** that prompts for an angle in degrees from the keyboard and then prints out neatly on the screen the sine, cosine and tangent of the angle.

An introduction to Fortran 90: PART 3

9. Arrays

So far in this course, you have dealt with variables which have one value. But this would be a difficult way to work with a vector or matrix, which could contain many values. Instead, vectors and matrices can be stored in *arrays*, which can contain many elements. Individual elements of the array are identified by the use of subscripts. For example, the position vector of a point in space *(x,y,z)* could be represented by the one-dimensional array **r**. Array **r** would contain 3 elements, with **r(1)** representing the *x* coordinate, **r(2)** the *y* coordinate and **r(3)** the *z* coordinate.

Fortran 90 allows arrays to have up to 7 dimensions. An array is declared in the normal way using a declaration at the beginning of the program, but both the type of array (*real*, *integer*, etc) and the dimensions of the array must be declared. So, for example, the 3-element vector above might be declared as

```
real,dimension(3)::r
```

This states that **r** is an array with three elements in one dimension. Since **r** is declared as a **real** array, each element of **r** will contain a **real** value. Similarly, a 2 by 3 integer matrix might be declared as:

```
integer,dimension(2,3)::mymat
```

and every element of **mymat** would be an integer. Note that the array **mymat** has 2 rows and 3 columns.

Note that it would also be possible to declare mymat as:

```
integer::mymat(2,3)
```

This format is more convenient when you wish to declare arrays of different sizes in the same declaration statement.

Fortran 90 allows you to perform calculations with an array in one of several ways:

- apply the same operation to all elements of the array, e.g. multiplying by 2.
- apply a single operation to the whole array, e.g. calculating the sum of the elements.
- perform a calculation on individual elements.
- perform a calculation with elements that match a certain condition, e.g. all elements with values greater than zero.

The following subsections describe these options.

9.1 Whole array elemental operations

Fortran 90 allows you to perform calculations on all the elements of an array at once using *whole-array elemental operations*. Nearly all intrinsic functions work on arrays, so if **a** and **b** are arrays with the same shape, then the following are all valid:

Statement	Resulting value of each element of array b
`b=a+2`	2 more than the corresponding element of **a**
`b=a-2`	2 less than the corresponding element of **a**
`b=a*2`	2 times the corresponding element of **a**
`b=a*c`	the corresponding element of **a** multiplied by the corresponding element of array **c**
`b=a/c`	the corresponding element of **a** divided by the corresponding element of **c**
`b=a**n`	the corresponding element of **a** raised to its n'th power
`b=1.0`	1.0
`b=a`	the corresponding element of **a**
`b=cos(a)`	the cosine of the corresponding element of **a**
`b=sqrt(a)`	the square root of the corresponding element of **a**

Type in the simple program below, which reads in the components of two vectors and then calculates the sum of the vectors. Call your program **vector1.f90**. Use some trial values to check that the program sums the vectors correctly.

```
program vector1
! Program to add two vectors
implicit none
real, dimension(3):: vect1, vect2, vectsum
!
! Read in the elements of the two vectors
!
write(*,*)' Enter the three components of vector 1: '
read(*,*) vect1
write(*,*)' Enter the three components of vector 2: '
read(*,*) vect2
!
! Now add the vectors together by adding their components
!
vectsum = vect1 + vect2
write(*,10) vectsum
!
10  format(' The sum of the vectors is: ', 3f6.2)
stop
end program vector1
```

9.2 Whole array operations

Some intrinsic functions work on arrays in a different way. For example, the **sum** function operates on a whole array to find the sum of all the elements. Unlike the functions that you have used so far, this function is specific to arrays. Some more intrinsic functions that are specific to arrays are:

Command	Example	Effect
product	**product(a)**	Product of all the elements in array *a*
sum	**sum(a)**	Sum of all the elements in array *a*
dot_product	**dot_product(v1,v2)**	Dot product of vectors *v1* and *v2*
matmul	**matmul(a,b)**	Multiply two conformal matrices *a* and *b*
maxval	**maxval(a)**	Maximum value in array *a*
minval	**minval(a)**	Minimum value in array *a*
maxloc	**maxloc(a)**	Array of location(s) of the maximum value in array *a*
minloc	**minloc(a)**	Array of location(s) of the minimum value in array *a*

Exercise

The length of a vector **r** is given by $\sqrt{r_1^{\,2} + r_2^{\,2} + r_3^{\,2}}$, where r_1, r_2 and r_3 are the components of the vector. Edit your program **vector1.f90** to find the length **rlen** of the vector **vectsum**. Call your new program **vector2.f90** and test it with some trial values.

Hint: try the functions **sqrt** and **sum**.

9.3 Working with subsections of arrays

9.3.1 Selecting individual array elements

If you wish to work with a single element of an array, refer to it specifically. For example, the three elements of vector **vect1** are referred to as vect1(1), vect1(2) and vect1(3), so the length of **vect1** could be calculated explicitly as:

```
length = sqrt( vect1(1)**2 + vect1(2)**2 + vect1(3)**2 )
```

When you work with multi-dimensional arrays, it is important to know which subscript refers to which dimension, e.g. when you multiply two matrices. The pattern of the elements in a 2 by 3 matrix **mymat** can be visualised as:

```
(1,1)    (1,2)    (1,3)
(2,1)    (2,2)    (2,3)
```

so the top left element would be **mymat(1,1)** and the bottom right one would be **mymat(2,3)**.

Note that array element numbering starts by default at element 1 (not zero). It is possible to assign a different range for the subscripts which identify the elements in the array. For example, in the above case a 2 by 3 integer array could also have been declared in the form:

```
integer,dimension(0:1,-1:1):: mymatrix
```

In this case the array is of the same 2 by 3 form but the range of the subscripts has been changed and the pattern of elements can now be visualised as:

```
(0,-1)    (0,0)    (0,1)
(1,-1)    (1,0)    (1,1)
```

and the array elements must now be addressed using the modified subscript indices. If you choose to reference your arrays in this way, you should be aware that functions such as **maxloc** give the *location* of an element, not the subscript.

Note: One of the most frequent problems with arrays is an *array bound error*. This occurs when a program attempts to read or write to an array element that does not exist, such as the 11th element of a 10-element array. Section 12 describes how to use the compiler to help avoid or correct such errors.

Exercise

Write a program **mat1.f90** that requests two 2x2 integer matrices from the keyboard and then multiplies them. Using references to individual array elements, display the input matrices and the result in the order:

```
(1,1)  (1,2)
(2,1)  (2,2)
```

Check that the ordering of elements in the matrices is as you expected and that the multiplication is correct for this ordering.

9.3.2 Selecting array sections

A subsection of an array can be selected by specifying a range of subscripts. For example, the selection **mymat(1:2,2:3)** would select these elements from a 4 by 4 matrix:

$$\begin{pmatrix} - & * & * & - \\ - & * & * & - \\ - & - & - & - \\ - & - & - & - \end{pmatrix}$$

If one end of the range is left out, the lower or upper bound of the array is used as a default. The following examples are valid subsections of arrays:

`mymat(3,3:4)`	Row 3, columns 3 - 4
`mymat(3,2:)`	Row 3, all columns from column 2 to the end column
`mymat(3,:)`	Row 3, all columns
`mymat(3:3,3:3)`	Single element: row 3, column 3.

9.3.3 Using masks

Sometimes you may not wish to apply a function to every element in an array. For example, it would not be appropriate to apply the **log** function to elements with negative values. Under these circumstances, use of a *mask* allows you specify which elements to include.

To illustrate use of a mask, consider the following statement:

```
where(a>0) b = log(a)
```

The **where** statement is used to perform an array operation only on elements where a certain logical condition is true. In the example above, **a** and **b** are arrays with the same shape. The beginning of the statement, **where (a>0)**, generates a *mask* of locations where the elements of **a** have values greater than zero. This mask is then applied to the rest of the statement. So the whole statement causes the operation **b=log(a)** to be performed for all elements of **a** that are greater than zero. Wherever **a** is zero or less than zero, the operation is not performed and the corresponding element of **b** is unchanged.

The condition **a>0** that forms the core of this example is a *logical expression*: the value of the expression is either **.true.** or **.false.**. Part 4 of this course gives more details of how to construct and use logical expressions to control the operation of your programs, including information on how to construct longer **where** statements.

Most array-based functions can use a mask directly. The examples in the table below are functions that make particular use of a mask:

Function	Examples	Description of function
`all`	**`all(a>1.0)`** **`all(a>0.0, dim=1)`**	Returns logical value **.true.** if the mask condition is true for all values (in given dimension, if specified).
`any`	**`any(a>=1.0)`** **`any(a>0.0,dim=1)`**	Returns logical value **.true.** if the mask condition is true for any values.
`count`	**`Count(a>0.0)`**	Number of values which match the mask condition.
`maxval`	**`maxval(a,dim=1)`** **`maxval(a,a<=0.0)`**	Maximum value in the given dimension or among elements that match the specified mask.
`minval`	**`minval(a,a>b)`**	As maxval, but gives the minimum value.

Exercise

Imagine that your department has asked you to write a program that records students' exam marks. Every student who scores more than 50% has passed the exam. You will need to use arrays to record the students'

names, their marks and whether they have passed. The department has also asked that you calculate the total number who have passed, the average mark, and identify the person with the top mark. Write a prototype program called **results1.f90** which processes results for 5 students and produces output similar to the following:

Student:	Fred	Susie	Tom	Anita	Peter
Mark:	64	57	49	71	37
Pass?	P	P		P	

No. of passes = 3
Average mark = 55.6
Prize awarded to Anita

9.4 Allocatable arrays

All the arrays that you have seen so far have been declared with a fixed size. Space for fixed-size arrays is allocated when a program is compiled and it cannot be changed without editing the program. Sometimes this can be inconvenient, for example when you do not know the size of the array in advance, or when you would like to re-use the memory space taken up by a large array that is used only for a short time within a program. Under these circumstances it can be more useful to use *allocatable* arrays.

The space for allocatable arrays is allocated in a manner similar to the way that files are opened and closed: you must **allocate** space for the array before you use it, and you must **deallocate** the space when you have finished with the array.

Since the size of an allocatable array is not known at the beginning of the program, its declaration gives the dimension of the array but not the size. Colons (:) are used to indicate the number of dimensions. The following statement declares an allocatable, 2-dimensional array:

```
real, dimension(:,:), allocatable:: myarray
```

Later on in the program, the size of the array is calculated in some way and space for the array is allocated. Notice that the size of each dimension must be an **integer**, e.g:

```
read(*,*) idim1, idim2
allocate(myarray(idim1,idim2))
```

Now that space for it has been allocated, the array myarray can be used as usual. Once it is no longer needed, the memory space is freed by the **deallocate** command:

```
deallocate(myarray)
```

Useful functions for working with allocatable arrays include:

allocated	**allocated(myarray)**	Returns logical **.true.** if the array is allocated.
shape	**dims = shape(myarray)**	Returns an integer array containing the sizes of each dimension in the array.
size	**length = size(myarray)** **len2 = size(myarray,dim=1)**	Returns an integer number which is the total number of elements in the array, or the number of elements in the specified dimension.

Some examples of allocatable arrays are given later in this course.

10. Parameters and initial values

If you have a program with many arrays, it can be difficult to remember why particular arrays have particular sizes. One way round this is to use *parameters* in your array declarations, as illustrated below:

```
program arrays
integer,parameter:: imax=100, jmax=200
real,dimension(imax):: array1
real,dimension(imax,jmax+imax):: array2
real,dimension(3):: array3
etc....
```

In this program fragment, definitions of some *parameters* are included among the declarations. The parameters are not variables because they cannot be changed later in the program. Notice that the value of the parameter is set in its declaration statement.

Once it has been declared, a parameter can be used wherever you would use a constant, including in array declarations as shown. This can help avoid errors if you edit your program later to accommodate different array sizes, because you need only make the changes in one place. Notice the position of the parameter declaration: it must come *before* the parameters are used.

Values for variables can also be set in the declaration statements. However, because the values of variables can be changed during operation of the program, these will be *initial* values only. The example above could have initialised the three arrays as follows:

```
Prog1ram arrays
integer,parameter:: imax=100, jmax=200
real,parameter:: pi=3.14159
real,dimension(imax):: array1=pi
real,dimension(imax,jmax+imax):: array2=0.0
real,dimension(3):: array3=(/1.5,2.0,2.5/)
etc....
```

All the elements in an array can be set to the same initial value as shown for **array1** and **array2**, while the syntax for entering several initial values is shown for **array3**. Notice that initial values can only be constant expressions.

Exercise

Edit your program **results1.f90** so that the number of students in the class is described as a parameter.

An introduction to Fortran 90: PART 4

11. Program control: DO loops and IF statements

A program normally executes statements in the order that they appear. That is, the first statement is executed first, and then each subsequent statement is executed in turn until the end is reached. Sometimes this will be too simple for your programming needs. In this section of the course, you will see how to repeat sections of your program with **do** loops and how to use **if** statements to choose whether to execute parts of your program.

11.1 DO... END DO loops

A simple **do** loop has the form:

```
do index=istart,iend,incr
     statement 1
     statement 2
     statement 3
     etc
end do
```

This **do** loop is contained between two statements: the **do** statement and an **end do** statement. All the statements between these lines are executed a number of times. On the first pass through the loop, the variable *index* has the value *istart.* After this first pass, *index* is incremented by an amount *incr* each time the program goes through the loop. When the value of *index* is greater than *iend* (or, if *incr* is negative, when *index* is less than *iend*) then the statements inside the loop are not executed and program execution continues at the line after the **end do** statement. If the increment *incr* is omitted then, by default, *incr* is assumed to be 1.

Note that *istart*, *iend* and *incr* may be positive or negative constants, variables or expressions, but they should always be **integers.**

Notice that the statements in the example **do** loop are indented by a few extra spaces. This is a good programming style to use. It helps you see easily where a loop begins and ends.

Warnings!

- Never use any of the statements inside the loop to alter the **do** statement variables *index*, *istart*, *iend* or *incr*.
- Never attempt to jump into the body of a **do** loop from outside the loop - this is illegal.
- Do not assume that *index* will have the value *iend* after normal conclusion of the **do** loop. Assume that it is undefined. If exit occurs from the **do** loop before the normal conclusion, e.g. by use of an **exit** statement (see later), then *index* will retain its value.

Exercise

Type the following program into a file called **roots1.f90** and then try running it with trial values of *n*.

```
program roots1
implicit none
integer:: i,n
real:: rooti
!
! Program to demonstrate the use of a DO loop.
!
write(*,*)' Enter an integer'
read(*,*)n
do i=2, 2*n, 2
    rooti=sqrt(real(i))
    write(*,*) i, rooti
end do
stop
end program roots1
```

This program reads in an integer (*n*) and then writes out the first *n* even numbers, together with their square roots. The **do** loop starts with the loop index *i* set to 2, then executes all statements down to the **end do** statement. In the first pass through the loop, the numbers 2 and $\sqrt{2}$ are displayed on the screen. At the beginning of the second pass *i* is incremented by 2 and takes the value *4*. As long as *i* is not greater than *2*n*, the statements will be executed again and the numbers *4* and *2* will be displayed on the screen. The loop will continue until the last numbers (2*n* and $\sqrt{2n}$) are displayed.

Exercise

Alter the program **roots1.f90** so that it displays the results in reverse order. Name the altered program **roots2.f90**.

Exercise

In part 3 of this course you wrote a program **results1.f90**, which recorded the exam marks of 5 students. The head of department is so pleased with your program that he would like a version that can handle large classes. Unfortunately, your original output layout is not suitable for larger numbers of students. Modify your program so that it uses a **do** loop to display the arrays in a vertical layout rather than a horizontal one, e.g:

Student:	Mark:	Pass?
Fred	64	P
Susie	57	P
Tom	49	
Anita	71	P
Peter	37	
No. of passes =	3	
Average mark =	55.6	
Prize awarded to	Anita	

Test the program by entering marks for a larger number of students.

Hint: you may not know how many students are in a class. To make sure that your arrays are big enough to handle a whole class, either use allocatable arrays or declare the arrays with the maximum dimensions that you think they could need.

11.2 IF statements

An **if** statement allows you to choose to execute a part of your program *if* a certain condition is true. The example below shows the general layout, which is similar to a **do** loop:

```
if (logical expression) then
    statement 1
    statement 2
    statement 3
    etc
end if
```

The **if** and **end if** statements enclose a block of statements, which should be indented to help you see where the **if** block starts and ends. The **if** statement contains a *logical expression* in brackets, which can have a result of either **.true.** or **.false.**. If the result of the logical expression is **.true.** then all the statements inside the block are executed. If the result of the logical expression is **.false.** the statements are not executed.

The logical expression in an **if** statement is constructed from a special syntax which uses *logical operators*. These are similar to normal operators but do not set the values of variables - they just compare them. Some examples of logical operators are given below, together with an illustration of their use:

Operator	Alternative (older) form	Example	Explanation
==	**.eq.**	if(i == j) then…	equals
>	**.gt.**	if(i.gt.j) then…	greater than
>=	**.ge.**	if(i.ge.j) then…	greater than or equal to
<	**.lt.**	if(i < j) then…	less than
<=	**.le.**	if(i<=j) then…	less than or equal to
/=	**.ne.**	if(i.ne.j) then…	not equal to
	.not.	if(.not. k) then…	**.true.** if **k** is **.false.** and **.false.** if **k** is **.true.**
	.or.	if(i>j.or.j<k) then…	logical **or**
	.and.	if(i>j.and.j<k) then…	logical **and**

Note that there are two forms of most of the logical operators. Use whichever you prefer. There are two forms because only the older form (e.g. **.le.**) was available in previous versions of Fortran and this form is still permitted as an alternative to the newer form.

A logical expression is evaluated in the following order:

1) Arithmetic operations are evaluated first and then followed by, in order:

2) **.eq.**, **.ne.**, **.gt.**, **.ge.**, **.lt.** and **.le.** have equal precedence and are evaluated from left to right. These operators are identical to: **==**, **/=**, **>**, **>=**, **<** and **<=**.

3) **.not.** operators.

4) **.and.** operators, from left to right.

5) **.or.** operators, from left to right.

As with arithmetic operations, you can use brackets to change the order of evaluation.

Warning! .eq. should NOT be used to compare the values of **real** or **double precision** variables. Real variables are held only to machine precision so, instead of using **.eq.** to see if they are equal, use **.ge.** or **.le.** to check whether the difference between two real numbers is small.

Logical variables need not be compared to anything, since they can only have the values **.true.** or **.false.** . If *mylog* is a logical variable, then it can be used in a logical expression as follows:

`if(mylog)`	succeeds if *mylog* is **.true.**
`if(.not.mylog)`	succeeds if *mylog* is **.false.**

A more complicated example of a logical expression is the following:

```
if (a.le.b.and.i.eq.n.and..not.mylog) then
     statements
end if
```

which identical to:

```
if (a<=b.and.i==n.and..not.mylog) then
     statements
end if
```

The block of statements inside the **if** statement will only be executed if the value of the variable **a** is less or equal to that of **b** and **i is equal to n** and the logical variable **mylog** has the value **.false.** (i.e. if **.not.mylog** = **.true.**). Note that two full stops are needed when two old-style logical operators are next to each other (**.and.** and **.not.** in this case).

An **if** statement can be extended to choose between two options by including an **else** statement:

```
if (logical expression) then
      first block of statements
   else
      second block of statements
end if
```

The choice can be extended even further by using one or more **else if** statements, e.g:

```
if (logical expression 1) then
      statement block 1
   else if (logical expression 2) then
      statement block 2
   else
      statement block 3
end if
```

Note: if there is only a single statement inside the **if** block, then a single line form of the **if** can be used, e.g:

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

The **then** and **end if** statements are not required in this case.

Exercise

To test the use of the **if** statement and logical expressions, try out the following program, **testif.f90**:

```
program testif
implicit none
real:: a,b
logical:: logicv
!
! Simple program to demonstrate use of IF statement
!
write(*,*)' Enter a value for real number A '
read(*,*)a
write(*,*)' Enter a value for real number B '
read(*,*)b
write(*,*)' Enter .true. or .false. for value of logicv '
read(*,*)logicv
!
if (a<=b.and.logicv) then
    write(*,*)'a is less or equal to b AND logicv is .true.'
else
    write(*,*)'Either a is greater than b OR logicv is .false.'
end if
!
stop
end program testif
```

Exercise

Modify your program **testif.f90** to include a **do** loop which runs through the test in the above program for five different sets of variables before coming to a stop. Don't forget to modify the indentation accordingly!

11.2.1 More about the where statement

Earlier in the course the **where** statement was introduced. This is similar to an array version of the **if** statement and also uses logical expressions to determine which actions to take. As for the **if** statement, there is a longer form of the **where** statement that allows you to perform several array operations under the same logical conditions. For example:

```
where(a>0)
      b = log(a)
      c = sqrt(a)
end where
```

and it is also possible to specify operations to perform when the condition is not true:

```
where(a>0)
      b = log(a)
      c = sqrt(a)
elsewhere
      b = -999.0
      c = -sqrt(a)
end where
```

Remember that the **where** statement is used with arrays, so **a**, **b** and **c** in this example are all arrays.

11.3 CASE statements

Repeated **if..then..else** statements can often be replaced by a **case** statement, for example:

```
select case season
case ('summer')
     leaves = .true.
     sun = .true.
case('winter')
     leaves = .false.
     sun = .false.
case('spring','autumn')
     leaves = .true.
     sun = .false.
case default
     write(*,*) 'mistake in season name'
end select
```

The **case** statement chooses which operations to perform, depending on the value of an expression that can be **integer**, **character** or **logical**, but not real or double precision. In the example above, the **case** statement depends on the value of the character variable **season**. Note that:

- For any value of **season** only one of the cases is executed.
- The statement **case('spring','autumn')** shows how to execute the same set of operations for any one of a list of cases.
- The **case default** is optional and will be executed if none of the other cases is matched.

11.4 Controlling DO loops with logical expressions

A simple **do** loop executes a fixed number of times. Sometimes this may not be appropriate so Fortran 90 allows some variations, which are discussed in this section.

11.4.1 Conditional exit loops

In these, the loop is terminated by a conditional **exit** statement, ie. an **if** statement which executes an **exit** action. Note that the loop can be infinite if the **exit** is never executed! This is a potential problem with all **do** loops that do not have a fixed number of iterations.

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

11.4.2 Conditional cycle loops

Conditional cycle loops are constructed in a similar way to the loop shown above, but the action of the **if** statement is to cause the program to cycle immediately to the next iteration of the loop, skipping any remaining statements in the current cycle.

```
do
   statements
   if (logical expression) cycle
   statements
end do
```

11.4.3 DO WHILE loops

A **do while** loop takes the following form:

```
do while(logical expression)
   statement 1
   statement 2
   statements
end do
```

This is very similar to the conditional exit loop. The statements in the **do while** block are executed for as long as the result of the logical expression remains **.true.**. It is easier to see this in the following example program, which you should type into the file **dowhile.f90** and run:

```
program dowhile
implicit none
real:: x
logical:: repeat
!
! Simple program to demonstrate DO WHILE loop.
!
repeat=.true.
do while(repeat)
    write(*,*)' Enter a real number x, '
    write(*,*)' or a negative number to exit. '
    read(*,*) x
    if(x.gt.0.0)then
        write(*,*)' The square root is ',sqrt(x)
    else
        repeat=.false.
    end if
end do
stop
end program dowhile
```

Exercise

Once you have run this program successfully, modify it so that if x<0 it calculates the square root of (-x) and so that it exits if $|x|<0.1$.

Exercise

Write a program called **quad.f90** to solve the quadratic equation $ax^2 + bx + c = 0$. The program should ask you to input 3 numbers *a*, *b* and *c,* then check whether the roots of the equation are real (i.e. check that $b^2 - 4ac \geq 0$) and then, if the roots are real, find them from the formula:

$$roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If the roots are not real, the program should write a message to say this. Test your program using a=1, b=-3, c=2 and then a=1, b=1, c=2.

11.5 Named DO loops and IF statements

When you have a large number of **do** loops and/or **if** statements in your program, it can sometimes be helpful to name them, e.g:

```
myloop:  do i=1,5
      x(i)=real(i)
end do myloop
```

and

```
fred:  if (y<x) then
      y=0
end if fred
```

Why is this useful? One reason is that naming a loop in a conditional **exit** statement or a conditional **cycle** statement allows you additional program

control. For example, if you have several nested **do** loops (that is, one inside the other), a conditional cycle statement in the inner loop could specify cycling either the inner loop *or* the outer loop. In the program fragment below, a loop over **i** encloses a loop over **j**. For each value of **i** the program will run through the loop in **j** until **j=i**. After that point it will skip the rest of the inner loop and cycle to the next iteration of the outer loop.

```
outer: do i = 1, 10
    inner:  do j = 1,10
        if (j>i) cycle outer
        statements
    end do inner
end do outer
```

11.6 Implied DO loops

A compact way of reading in (or writing out) the elements of an array is by use of an **implied do** loop, which is a structure available only for dealing with input/output. A simple example is to read in the 3 elements of array **r**:

```
read(5,*)(r(i), i=1,3)
```

This statement is a shorthand way of telling the program to read in **r(i)** with **i** taking the values 1 to 3. The structure **i=1,3** here has the same meaning as it does in any normal **do** loop. So the program will read **r(1)** followed by **r(2)** and finally **r(3)**.

Of course, a loop is not necessary to read in such a simple array. However, for the 2 by 3 array **mymat**, we could use a more complicated implied **do** loop to write out the array elements in the order **mymat(1,1)**, **mymat(1,2)**, **mymat(1,3)**, **mymat(2,1)**, **mymat(2,2)**, **mymat(2,3)** using:

```
write(6,20)((mymatrix(i, j), j=1,3), i=1,2)
20  format(3i6)
```

When several implied **do** loops are nested within each other, each loop is enclosed in its own pair of brackets. The innermost loop (the **j** loop in the example above) is done first.

Exercise

In section you wrote a program called **mat1.f90**. Edit **mat1.f90** so that it prints its output using an implied **do** loop. Your output should appear in the actual pattern of the array.

12. Hints on debugging programs

Typing errors are a common programming problem. In the following program, the variable *x1* has been mis-typed on line 5 as *x11* and there is no **implicit none** statement to trap this. So, because of Fortran's implicit typing, *x11* is accepted as a **real** variable when the program is compiled. The program's output will be wrong but there is no clue to tell you why.

```
program typo
real:: x1, y1
write(*,10)
read(*,*) x1
y1 = x1 **2 + 2*x11 + 1
write(*,20) x1, y1
10 format('Enter a number: ')
20 format('(',f6.2,' + 1) squared is',f6.2)
end program typo
```

This error could have been avoided by using an **implicit none** statement. An **implicit** statement should go on the second line of the program, immediately after the program name. It will tell the compiler not to accept any variables that have not been declared explicitly.

There are other forms of the **implicit** statement that you can use to override Fortran's implicit typing, although they are not recommended. For example, you could change the default type to double precision for all variables with beginning with the letters *s* to *z*, by using:

```
implicit double precision (s-z)
```

If you are looking for a more subtle error in a program, it will often help to add some **write** statements to output the values of variables at various points in the program. This can help you find which part of the program has a problem and which variables are affected. Some simple things to look for are:

- Integer division
- Division by zero - this can lead to a 'Not a Number' (NaN) error.
- Problems in mixed-type calculations (e.g. integer and real)
- Integer overflows (i.e. integers that are too large for the **integer** type)
- Floating point overflows (numbers that are too large for **real** type)
- Loss of significance, e.g. when one long number is subtracted from another.
- Brackets in the wrong place
- Accidental use of the letter l instead of number 1, and letter O instead of number 0. (Do not have variables called O or l!)
- Incorrect declarations, e.g. array variables not declared as arrays
- Use of reserved function names for variables (e.g. sqrt, sin, etc).

You should also avoid giving your programs names that duplicate operating system commands. For example, 'test' and 'sort' are both Unix commands and may be run instead of your programs with those names. If you suspect that this could be happening, type e.g:

```
which test
```

To run your program explicitly, type the path to it when you run it, e.g:

```
~/Fortran/test
./sort
```

Arrays are a very frequent source of problems in programs. Perhaps the most frequent type of error is that the program refers to an array element that does not exist, e.g. the 11th element of a 10-element array. This is called an *array bound error.* The program might still run when this happens (the results will probably be wrong), or the error might cause the program to crash, perhaps with a *segmentation fault*.

The compiler does not check for array bound errors unless you include the option -C, e.g:

```
pgf90 -C -o myprogram myprogram.f90
```

If this option is included, the compiler will generate warnings during compilation if it detects any array bound errors. It will also cause the program to stop and generate an error message if an array bound error occurs during execution.

The compiler does not include array bound checking by default because it slows down the compilation process and increases the size of the executable file. It might also slow down the program's execution. However, including the **-C** option can make it much easier to find array-bound errors in your programs.

The man page for **pgf90** also includes information about the many other options that can be used. Notice that the options **-C** (array bound checking) and **-c** (compile only, do not link) are different!

Compiling with the **-g** option allows you to use a *debugger* program such as **pgdbg**, to debug your program. **pgdbg** is a powerful tool for finding bugs in large or complicated programs but it also takes some effort to learn. Although this course does not cover it, you should consider learning **pgdbg** if you will be working on large programming projects. (The debugger on the ITS Suns is called **dbx**.)

An introduction to Fortran 90: PART 5

13. Subprograms

When you use the intrinsic functions *sin, sqrt*, etc, you are using examples of subprograms. The intrinsic functions are included as part of Fortran, but you can also write your own subprograms. Subprograms are independent blocks of Fortran which perform a specific task. They may be written and compiled separately, and are joined to a main program when the program is linked. In large programming projects, subprograms may be written by different people.

One of the main advantages of subprograms is that they may be written and tested as isolated units to ensure that they operate correctly. This makes the task of developing a large program much easier because it can be split into a number of smaller sections. A subprogram is normally tested by writing a simple main program which provides some test data and displays test results. This way, any errors or "bugs" can be confined to a relatively small section of code and are much easier to track down and correct.

Another advantage of subprograms is that they are portable. Once you have a working, tested subprogram, you can use it in later programs whenever you need it. This can greatly reduce the effort required in any large program development project.

We will consider two different ways of constructing subprograms:

- *Functions* are used by referring to their name (exactly as you would refer to the intrinsic function *sin(x)*) and they produce one answer.
- *Subroutines* are a more general form of subprogram. They can perform complicated tasks that return one or more answers. Alternatively, a subroutine that generates a file or some graphics might not return any answer to the main program.

13.1 Functions

Functions are used to generate a single answer. They should not have "side-effects", such as writing to the screen, and they usually carry out relatively simple tasks. A function is self-contained, so it could easily be copied for use in other programs that need it.

Earlier in the course you wrote a program **quad.f90** that solved a quadratic equation. The example program below, **fn1.f90**, is very similar: it finds the real roots of a quadratic equation, provided that real roots exist, and displays the larger root. The example consists of a main program and one function subprogram, called *bigroot*.

The function *bigroot* tests to see if the roots are real and returns the bigger one. If the roots are imaginary, *bigroot* returns an unrealistically large negative result. The **f12.6** format used by the main program to display the

result cannot cope with such a number and therefore a field of *********'s will be printed to indicate that no real root has been found.

```
program fn1
implicit none
real:: a,b,c,bigroot
!
! Program to demonstrate the use of a FUNCTION subprogram
!
write(6,10)
read(5,*) a, b, c
write(6,20) bigroot(a, b, c)
10  format(' Enter the coefficients a, b, c '/)
20  format(' The larger root is ',F12.6)
stop
end program fn1
!
! End of main program
!
function bigroot(a, b, c)
implicit none
real:: bigroot, a, b, c, test, root1, root2
!
! Function to find largest root of a quadratic
! If no real roots then function returns value -9.0E35
!
test = b*b - 4.0*a*c
if(test.ge.0.0) then
   root1 = (-b + sqrt(test)) / (2.0 * a)
   root2 = (-b - sqrt(test)) / (2.0 * a)
   if(root2.gt.root1)then
      bigroot = root2
    else
      bigroot = root1
   end if
 else
   bigroot = -9.0e35
end if
return
end function bigroot
```

Exercise

Modify your program **quad.f90** so that it uses functions to find the roots of the quadratic equation.

Note the following points about functions:

- If the function is written in the same file as the main program, it must appear *after the end* of the main program.
- The function must begin with a line that defines it as a function and lists the *arguments*, e.g:

```
function bigroot(a,b,c)
```

When you use the function to calculate a result, you must give arguments of the correct type and in the correct order.

- There must be an **end** statement at the end of the function definition in order to tell the compiler where the subprogram comes to an end. The statement can take one of the following forms:

```
end
end function
end function function_name
```

- Before the **end** statement there must be a **return** statement. At this point control is returned to the main program.
- The function should be declared in any (sub)programs that use it. Function *bigroot* returns a **real** answer, so it is declared as a **real** function on line 3 of program **fn1**.
- The declaration statements in the function generally obey the same rules that apply in a main program. (One exception relating to arrays is discussed in the notes on subroutines, later in this document.)
- The function definition must include the type of the value returned by the function, e.g. **real**, **integer** etc. This can be defined as above in a declaration statement within the function:

```
real:: bigroot
```

or alternatively in the first line of the function definition:

```
real function bigroot(a,b,c)
```

- In order to ensure that a value is returned by the function there must be at least one assignment statement in the function definition that assigns a value to the function. In the example program **fn1.f90** there are three such assignments depending on the values of **a**, **b** and **c**: **bigroot = root1**, **bigroot = root2** and **bigroot = -9.0e35**.
- The variables listed in the argument list in the function definition are called *dummy* variables. Their names do not have to match the names used in parts of the program where the function is called. You can confirm this in your example program by changing all occurrences of **a**, **b** and **c** *in* the function to **p**, **q** and **r** and re-running the program.
- It is *possible* to change the value of a variable in the argument list by an assignment statement within the function subprogram, but a well designed function should *never* be allowed to modify any of the arguments supplied.
- Apart from the variables passed as arguments, all of the variables within the function are entirely local to the subprogram. This means that identically named variables may exist in other programs and subprograms but these are completely distinct and not related in any way.

Exercise

Write a main program called **factor.f90** which reads in an integer *n* from the keyboard and then uses an integer function to calculate *n!* (the factorial of *n*). The main program should display the result on the screen. Reminder: *n! = n*(n-1)*(n-2)*….*3*2*1.*

13.2 Subroutines

You have seen above how functions may be used to calculate a single value. In contrast, a subroutine is used when a more general computation is required, possibly involving the return of several values of different types. Alternatively, a subroutine may not return any values as such. It may instead perform some other operation, such as displaying graphical results on the screen.

The following example program uses a simple subroutine called **solvit** to find both roots of a quadratic equation. The subroutine is invoked or 'called' by the **call** statement in the main program:

```
call solvit(a,b,c,root1,root2,realroots)
```

It is common for the main program of a large programming project to contain relatively few executable statements and a large number of calls to subroutines, making it easier to understand the purpose and structure of the program.

```
program subrout1
implicit none
real:: a,b,c,root1,root2
logical:: realroots
! demonstration of simple subroutine
write(*,10)
read(*,*) a,b,c
!
call solvit(a,b,c,root1,root2,realroots)
!
if (realroots) then
   write(*,20) root1,root2
 else
   write(*,*) 'Sorry, there are no real roots'
endif
!
10 format('Enter 3 coefficients')
20 format('The roots are',2f12.6)
stop
end
!
!subroutine solvit
!
subroutine solvit(a,b,c,root1,root2,realroots)
implicit none
real::a,b,c,root1,root2,test
logical::realroots
!
test=b**2 - 4*a*c
!
if(test>=0.0) then
  root1 = (-b + sqrt(test))/(2.0*a)
```

```
  root2 = (-b - sqrt(test))/(2.0*a)
  realroots = .true.
 else
  realroots = .false.
end if
!
return
end
```

Note the following rules for subroutines:

- When a subroutine is in the same file as the main program, it must appear after the end of the main program.
- Each subroutine must begin with a line that defines it as a subroutine and lists the arguments, e.g:

```
subroutine(orig,sorted,N)
```

- The subroutine definition must finish with an **end** statement. The statement can take one of the following forms:

```
end
end subroutine
end subroutine subroutine_name
```

- Before the **end** statement there must be a **return** statement. At this point control is returned to the main program.
- Because a subroutine may return several different types of value or no value at all, it is not declared as having a specific type, e.g. **real**, **integer,** either within the main program or the subroutine itself.
- The declaration statements within a subroutine obey the same rules that apply to a main program, apart from the following exceptions for arrays.
  - Although you must explicitly declare the dimensions of any array that is *local* to a subroutine, you do not need to give an explicit declaration [e.g. *real,dimension(100)::orig*] of the dimensions of an array that is passed to the routine as an argument. Instead, pass the dimensions of the array in the argument list. (e.g, *real,dimension(N)::orig* is used in the example above.) This *assumed size* declaration is the simplest way to declare arrays in subprograms.
  - If you use this method to pass a multi-dimensional array to a subprogram, make sure you also pass the full dimensions of the array (e.g. *imax*, *jmax*) and use these to declare the array in the subprogram. If you wish to pass a section of an array to a subprogram, pass it in the form array(i1:i2,j1:j2) and pass the dimensions of this section as well. If you pass the *whole* array but want to use only a section, pass the dimensions of this section separately (e.g. *ifill*, *jfill*). This precaution is necessary because arrays are in fact stored in 1-D format, so the subroutine needs to read the first *ifill*, skip the unused elements up to *imax*, read the next *ifill,* skip to *2*imax*, and so on. If you did not give the full dimensions, the subroutine would simply

pick the first *ifill*jfill* elements in the 1-D list. A carefully written subroutine that uses this way of passing an array might look like this:

```
subroutine careful(arrayname,maxrow,maxcol,nrows,ncols)
real arrayname(maxrow, maxcol)
integer maxrow, maxcol, nrows, ncols
do i = 1,nrows
   do j = 1, ncols
      arrayname(i, j) = arrayname(i, j) + i * j
   end do
end do
end subroutine careful
```

A second way to deal with arrays in subprograms is to declare them as assumed shape arrays. Like allocatable arrays, these are declared with their dimensions but not their size, e.g:

```
real:: m1(:,:)
```

The subprogram will then know from the declaration statement that the array has 2 dimensions, but the array is left free to assume the shape of any array that is passed to it via the argument list. Assumed-shape arrays are easier to use if the subprograms are in modules and an example is given in section 14.

Subroutines and functions have some further features that you may find useful when planning and writing a program:

It is permissible for a subroutine or function to invoke itself. This is called recursion. A recursive subprogram must have the keyword recursive in its first line, e.g:

```
recursive subroutine mysub(a,b,c)
real recursive function myfn(x,y,z)
```

The arguments to subprograms can be declared in the subprogram with an *intent*. The intent of a variable in a subprogram can have one of three values: **in**, **out** or **inout**, e.g:

```
subroutine mysub(a,b,c)
real,intent(in)::a
real,intent(inout)::b
integer,intent(out)::c
```

A variable with an intent of **in** is passed to the subprogram but cannot be changed by it. One with an intent of **out** must be produced by the subroutine. One with an intent of **inout** is passed to the subroutine *and* can be changed by it. Use the **intent** attribute to help make sure that variables cannot be changed accidentally in a subprogram.

Exercise

Earlier in this course you wrote a program called **mat1.f90** to multiply two matrices together. Using this as a basis, write a program called **inverse.f90** that reads in the elements of a 2 by 2 **real** matrix and then uses a subroutine

to compute the inverse of the matrix. The inverse of matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is $\frac{1}{det}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$ where the determinant *det* is given by *a*d - b*c*.

Make sure that you check that the original matrix is not singular (i.e. ensure that the determinant is not equal to zero before you compute the inverse). Check the operation of your subroutine by multiplying the two matrices together and displaying the result.

13.2.1 Generating random numbers

Fortran 90 has two built-in subroutines that provide a facility for generating random numbers. The subroutines give programmers a portable way to generate random numbers, although the precise implementation details do depend on the compiler.

The actual random number is generated by a line

```
call random_number(mynum)
```

which generates a single random number or an array of random numbers (*mynum* in the example), drawn from a uniform distribution between 0.0 and 1.0.

The call to **random_number** is usually preceded by a line:

```
call random_seed(put=iseed)
```

which initiates the random number generation with an integer "seed" array (*iseed* in the example). The seed array often has only one element but must still be declared as an array, e.g:

```
integer,dimension(1)::iseed
```

If **random_seed** is not called, or if the seed is not changed, the series of random numbers will always be the same. This is useful for repeating program runs.

Exercise

Write a program called **ranguess.f90** which asks for a seed number, generates a random number in the range 0 to 10 and then prompts you to guess the value. The program should then tell you whether your guess is too high or too low and ask for a new guess. The program should stop when you get within a given margin (e.g. 0.2) of the actual random number. In case you get fed up with guessing, the program should also stop if your guess is negative.

Optional exercise (long)

If you have time, write a program called **mysort.f90** which reads a number N from the keyboard, checks that N is between 2 and 100, then calculates N random numbers in the range 0.0 to 1.0 and stores them in an array (remember that **random_number(***x***)** can generate an array *x* of numbers). The program should then sort these numbers into descending order and finally display the sorted numbers on the screen using an implied do-loop. Use one subprogram to handle the input of N and another to sort the array of random numbers.

(Hint: the sort can be done by searching through the array of N numbers and noting which array element contains the largest number. Begin by assuming that the first number in the array is the largest. When you have found the largest number, swap it with the number in the first array element. Next search through the array elements from 2 to N to find the second largest number etc until only the smallest number remains in the N'th element of the array. You will need a pair of nested **do** loops to do this.)

13.3 Storing subprograms in separate files

So far, your programs have always been in one file, which has included the main program plus any functions and subroutines that are needed. When you write larger programs, this can make the source code difficult to navigate and to manage, so it is normal to keep subprograms in separate files, even in separate directories. For example, if you completed the last exercise, your final version of **sortit.f90** should contain three parts: the main program, a subroutine to read *N* and a subroutine to sort the random numbers. These could have been stored in three files, e.g:

sortit_main.f90
readN.f90
sortarray.f90

You would need to compile all of these and link them together to make your executable program. The simplest way to do this would be to list them in the compilation command, e.g:

```
pgf90 -o sortit sortit_main.f90 readN.f90 sortarray.f90
```

This will work, but it will compile every component every time you use it, even if you change only one subprogram. It would also be tedious to type in the list for a program that involves many files. To speed up re-compilations and to save typing, you can compile and link in two separate stages. In the first stage, the source code is compiled to form object files, using the **-c** option (compile only) to tell the compiler not to link the files.

```
pgf90 -c sortit_main.f90 readN.f90 sortarray.f90
```

This will give three object files whose names end in **.o**. To link them together to form an executable, you would type:

```
pgf90 -o sortit sortit_main.o readN.o sortarray.o
```

If you now edited one of the files, you would need only to re-compile that one, and then link the object files as above. This can make it much easier to develop and test components of a program.

Compilation and linking of multiple files is managed most comprehensively by a **makefile**. A **makefile** allows you to define all the components of a program, the way that they depend on each other, and the compilation options and environment that you want to use. When the **make** command runs, it will use this information to identify and re-compile any parts of the program that have been modified. Makefiles are extremely valuable for large programming projects, for programs that are worked on by several people, and for when you may come back to work on a program after a long time. ITS Guide 176 gives an introduction to makefiles and the **make** command.

13.4 Using subroutine libraries

So far in this course you have used subprograms that were intrinsic to Fortran 90 or that you have written yourself. This section outlines some ways in which you can make use of collections of subprograms, or *libraries*, that have already been developed by other people to solve common mathematical problems efficiently and robustly. Using these external subprograms can save a lot of development effort and can help your programs run faster and work more reliably.

When you compile and link a Fortran program, you link with a standard set of subprogram libraries. You can also specify extra libraries to use by including the **-l** option (letter l, not number 1) and the name of the library in the **pgf90** command, e.g.

```
pgf90 -o progname -llibraryname  progname.f90
```

How does the compiler know where to find the libraries? Your UNIX account has a number of settings called *environment variables*, including one called **LD_LIBRARY_PATH**. To view your LD_LIBRARY_PATH, type:

```
echo $LD_LIBRARY_PATH
```

Each of the directories in this path contains libraries for various purposes. If you have problems linking with a particular library or get errors at link-time about undefined symbols, it may be that you need to include the option **-l libraryname** in your compilation command, and/or you may need to include the appropriate directory in your LD_LIBRARY_PATH.

Note: if you are working on the ITS Unix service (not the linux service), you will need to add the Fortran 77 compatibility library to your compilation command for any libraries that were compiled with F77, e.g:

```
f90 -o progname -lf77compat -llibraryname  progname.f90
```

13.4.1 The NAG library

The Numerical Algorithms Group (NAG) subroutine library is widely used because its large set of mathematical subprograms is well tested and robust. The library is documented in a **man** page and in a large set of manuals which are currently kept on the bookshelves in CM131. The library is also documented in the NAG web pages – refer to local copies at **http://www.dur.ac.uk/its/local/nag/**. An important point to remember is that real variables used with the NAG library must be declared as double precision.

Although the NAG library is extremely useful, it can be difficult from the manuals to work out how you should use it. Once you have identified the NAG routine that you want to use, it is often easier to use the **nagexample** command to generate an example program for the routine in question.

For example, there are several random number generators in the NAG library, described in chapter G05 of the manuals. One of the simpler random number routines is called **g05caf**. To generate an example program that calls this routine, type the following command. (Note that the name of the routine must be in lower case and the second character is a zero, not the letter 'O').

```
nagexample g05caf
```

This will generate a small test program in your current directory, that you can compile, link and run. The test program will be called **g05cafe.f** - the name of the routine, plus an '**e**' to show that this is an example program, and **.f** instead of **.f90** because the example is written in Fortran 77. Have a look in this file to see how the routine **g05caf** is used. A simplified version using f90-style syntax might look like this:

```
!G05CAF Example Program Text
!     Mark 20 Revised.  NAG Copyright 2001.
!     .. Parameters ..
integer,parameter:: nout=6
!     .. Local Scalars ..
double precision:: x
integer::i
!     .. External Functions ..
double precision g05caf
!
!     .. Executable Statements ..
write (nout,*) 'G05CAF Example Program Results'
write (nout,*)
call g05cbf(0)
do i = 1, 5
   x = g05caf(x)
   write (nout,99999) x
end do
stop
99999 format (1x,f10.4)
end
```

You should not find it difficult to translate between the two languages.

To re-compile the program, you will need to tell the compiler to link with the NAG library. Do this by including the option **-lnag** . Also include **-g77libs** because the NAG libraries were compiled with a different Fortran compiler called g77, and **-Mllalign** to lay out double precision variables in memory on their natural (8-byte) boundaries rather than the default 4-byte boundaries.

```
pgf90 -o g05cafe -Mllalign -g77libs -lnag g05cafe.f
```

or, for Sun Unix users:

```
f90 -o g05cafe -dalign -lnag -lf77compat g05cafe.f
```

Note: you can use the compiler to compile F77 programs or a mixture of F90 programs with F77 subroutines. The command will recognise a program's language as Fortran 77 from the **.f** filename extension.

Note: NAG are developing a Fortran90 version of the NAG library and this is available on the ITS Sun Unix service, though not the Linux service. It does not yet have equivalents for all of the elements in the Fortran 77 library, but most are available. For more information on the NAG libraries available in Durham, see:

http://www.dur.ac.uk/its/local/nag/

Exercise

The program **g05cafe.f** always generates the same series of numbers. This behaviour is set by the call to **g05cbf**, which sets the seed for the random number generator. To generate a different series each time you run the program, replace the call to **g05cbf** with a call to **g05ccf**. Compile and run the altered program. Check that it generates a new set of numbers each time the program is run. You may find it easier if you change the program into F90-style syntax or to read the comments on F77 syntax in section 15.

13.4.2 Other external libraries

Programs that involve solving intensive linear algebra problems or other numerically intensive work may be substantially improved by use of the online repository of mathematical software and reference papers at Netlib,

http://www.netlib.org/ or

http://www.mirror.ac.uk/sites/netlib.bell-labs.com/netlib/

You may download and use material from this web site. A subset of the most popular libraries has been collected by Sun and made available for Sun computers as the Sun Performance Library. For more information on the Sun Performance Library and how to use it, visit

http://docs.sun.com/

and search for the Sun Performance Library User's Guide.

13.4.3 The 'Numerical Recipes' book

NAG is commercial software and requires a licence, so it is not suitable if you need to share your program with a site that does not have the NAG libraries. If you do need to do this and the Netlib routines do not solve your problem, consult the Numerical Recipes book, available from the University Library and the ITS Helpdesk. This book contains descriptions of ways to handle numerical problems and also includes ready-written subprograms. The routines in Numerical Recipes are supplied as source code, not in a compiled library, so you can see exactly what they do and adapt them to your own needs. They may not, however, be as thoroughly tested as the NAG routines.

The Numerical Recipes book is also available on-line at **http://www.nr.com/**. If you do not want to type in the routines, they can be downloaded from this site but there is a charge.

Optional exercise

If you have time, write a program to:

- Generate N random numbers and write them to a file,
- Sort the numbers,
- Calculate and display the mean of the numbers,
- Calculate and display the median,
- Write the mean and median to the end of the data file.

Use subprograms to structure your program and consider which steps might have a solution in the NAG library or in Numerical Recipes.

14. Modules

Modules are a powerful concept in Fortran 90 and are particularly useful for programming projects that are large or that involve several people. They allow you to group subprograms and variables together and they give you additional control over the way that variables are used. Some of the main uses for modules are:

- To provide a central location for declarations of constants and variables that are used in several parts of a program, so that the declarations need not be repeated in numerous subprograms or passed in argument lists.
- To group a set of subprograms together.
- To define new types of variable and functions that work on these types. (These 'derived types' of variable are not discussed in this course.)

The general form of a module is:

```
module modulename
  data definitions and declarations
  ...
contains
  module subprogram definitions
  ...
end module modulename
```

The following rules apply to modules:

- If a module is contained in the same file as the main program, it must come *before* the main program.
- A module begins and ends with the lines:

```
module modulename
end module modulename
```

- Any program unit that makes use of the module's contents should include the following statement immediately after the program name:

```
use modulename
```

- After this line, the variables and subprograms contained in ***modulename*** can be used in the program unit. Notice that it is not necessary to declare the module variables in the main program. The **use** statement is sufficient.
- If the name of an item that is accessed from a module conflicts with another name from elsewhere, the **use** statement can include an instruction to use a different name locally. For example, if a module variable is called *modvar*, it could be used with name *newvar* in another program unit:

```
use modulename, newvar=>modvar
```

- If only certain items are needed from the module, these can be named in the **use** statement too:

```
use modulename, only: var1, var2, …
```

14.1 Sharing variables and constants

Suppose that you have a program that uses numerous subroutines and functions. Several of these program units use a common set of constants and variables. There are two ways in which this set of items can be shared:

- the items could be declared in every subprogram that needs them and the values could be passed between subprograms using argument lists, or
- the items could be declared in a module and each subprogram that needs them could declare that it will **use** the module.

The second method has the advantage that any changes can be done in a single module instead of many subprograms, so there is much less risk of error.

The example below shows a very simple module that defines the constants π and e, and a program that uses the module.

```
module constants
  implicit none
  real,parameter:: pi=3.1415927, ee=2.7182818
end module constants
!
program useconst
use constants
implicit none
real::radius,area
radius = 10.5
area = pi * radius**2
write(*,*) area
end program useconst
```

Although this example shows a module that is in the same file as the program that uses it, modules (and single subprograms) will typically be stored and compiled separately and linked with one or more programs. One advantage of this is that if you modify a small part of a large program, only the modified part need be re-compiled.

Exercise

Type the module definition above into a file **constmod.f90** and the main program into a file **const.f90**. Compile the module with the command:

```
pgf90 -c constmod.f90
```

The **-c** option causes the file to be compiled but not linked; it will result in a file **constmod.o**. You will also gain a **.mod** file for each module, so you will have a file called **constants.**mod.

Now compile the main program with the command:

```
pgf90 -c const.f90
```

Finally, once the two parts have compiled successfully, link them together with the command:

```
pgf90 -o const constmod.o const.o
```

Run the program **const** to check that it operates correctly.

What happens if you omit the line **use constants** from your main program and re-compile?

Note: you *can* compile and link both files in a single command but this will always recompile both parts:

```
pgf90 -o const constmod.f90 const.f90
```

The order of the arguments in this command is important: the module(s) should come first.

14.2 Module subprograms

The next example shows a longer module **simplestats** that contains functions to calculate the mean and standard deviation of a 1-dimensional array. The module also contains two *saved* variables. These are variables whose values are saved (i.e. remembered) between one call to a module subprogram and the next call. In this example, the saved variables count the number of times that each function is called. Saved variables are a good way to share information between program units.

```
module simplestats
implicit none
integer,save::mean_times=0, stdev_times=0
!
contains
!
! function to calculate means
real function mean(vec)
   real,intent(in), dimension(:):: vec
   mean = sum(vec)/size(vec)
   mean_times = mean_times + 1
end function mean
!
! function to calculate standard deviations
real function stdev(vec)
   real,intent(in),dimension(:):: vec
   stdev = sqrt(sum((vec - mean(vec))**2)/size(vec))
   stdev_times = stdev_times + 1
end function stdev
!
end module simplestats
!
!!! main program starts here !!!
program usestats
use simplestats
implicit none
real, allocatable, dimension(:):: mydata
integer num, ist
! Input data
write(*,*) 'How many numbers are in the vector?'
read(*,*) num
! allocate array
allocate(mydata(num), stat=ist)
if (ist .ne. 0) then
 ! allocation failed
 write(*,*) 'Error allocating array!'
else
 ! enter numbers and calculate stats
 write(*,*) 'OK, enter the numbers...'
 read(*,*) mydata
 write(*,*) 'The mean is ', mean(mydata)
 write(*,*) 'The standard deviation is ', stdev(mydata)
 deallocate(mydata)
end if
```

```
! usage stats from saved module variables.
write(*,*) 'Calculated ',mean_times, ' means and ', &
     stdev_times,' standard deviation.'
end program usestats
```

Notice that the functions in **simplestats** are *internal* to the module. They come after the line:

```
contains
```

and before the end of the module. Internal functions and subroutines *must* end with the statement **end function** or **end subroutine**, not just **end**.

Putting your subprograms into modules helps you avoid using the wrong type of variable when you refer to a subprogram. Normally Fortran will allow you to pass, for example, an integer variable to a subprogram that requires a real number. The program will compile, but will probably not work correctly and the error will be hard to find. If the subprogram is in a module, however, the compiler will automatically construct an *interface* block to describe how the module subprograms should be used, so that attempts to pass the wrong type of variable in an argument list will be rejected.

Exercise

Modify your program **constmod.f90** to include a module subprogram that calculates the circumference of the circle.

As in the previous exercise, remove the line **use constants** and re-compile to see what effect this error has.

15. About Fortran 77

You may come across many Fortran programs that were written in Fortran 77. The aim of this section is to help you understand, use and perhaps update them.

Fortran 90 includes the whole of Fortran 77. However, some language features that were common in Fortran 77 should no longer be used. In addition, you may find programs that use non-standard, machine-specific extensions to the Fortran 77 language.

'Obsolescent' parts of the language which are permissible in Fortran 90 but not in Fortran 95 include:

- **real** and **double precision do**-loop index variables
- shared **do**-loop termination
- branching to an **end if** from outside the **if** statement
- alternative **return** statements
- **pause** statements

- **assign** statements
- assigned **goto** statements
- assigned **format** statements
- arithmetic **if** statements
- **H** format (hollerith format)

'Undesirable' features are permissible in Fortran 90 and in Fortran 95, but their functions can be performed in other ways. They include:

- "fixed form" syntax (see section) - use free form instead
- implicit declarations of variables - always use **implicit none**
- **common** blocks - use modules instead
- **computed goto statements**
- **equivalence** statements
- **entry** statements
- assumed size arrays - use assumed shape.

Try to remove obsolescent and undesirable features from old programs wherever possible.

15.1 Fixed form syntax

Prior to Fortran 90, Fortran required a very specific syntax. Each line of an old-style (usually Fortran 77) program was interpreted in the following manner:

Lines in a Fortran 77 program could be up to 72 characters long. If a line was longer than this, the characters beyond column 72 were ignored.

Columns 7 to 72 contained the body of the Fortran 77 statement. Any blanks were ignored and could be inserted at will in order to improve the layout of the statement.

Columns 1 - 6 were reserved for special purposes, described below:

- If column 1 contained the letter **C**, ***** or **!** then the line consisted of a **comment** statement.
- If a number from 1 to 99999 appeared anywhere in columns 1 to 5 then the number was a **statement label**.
- Column 6 was normally empty but if a character (**&** for example) did appear in this position then the line was a continuation of the previous one.

Errors were commonly encountered in Fortran 77 as a result of failing to comply with these rules.

15.2 Specific intrinsic functions

In Fortran 90, the same intrinsic function can be used for different types of argument. For example, **y = sin(x)** would be valid for real **x** and **y** and for double precision **x** and **y**. In Fortran 77 this was not the case: **sin(x)** would

be correct for real **x and would give a real answer**, but **dsin(x)** would have been used for double precision **x (and would give a double precision answer)**. Similarly, there are intrinsic functions called **dcos**, **dasin**, **dsqrt**, etc. In fact these are still used, but in Fortran 90 it is not usually necessary to specify them in the source code.

15.3 Common blocks

Common blocks were a way of sharing information between program units in a way that avoided lengthy lists of arguments. Instead of declaring each argument in each subroutine call, the information was put into common storage areas called **common blocks**. Although **common blocks** made it much easier to develop large programs, they were a frequent cause of problems and they can now be replaced by **modules**.

A common block is declared at the beginning of a program or subprogram, after the declaration of variables and before any executable statements. For example:

```
integer inum, jnum
real x, y, z, array1(10)
common inum, x, array1
```

The declaration of the common block began with the word **common** and was followed by a list of variables. In the example above, the common block contains an integer, a real number and a 10-element real array. Other program units could refer to this common block by declaring it in the same way, including the list of variables. The variables need not have the same names in all of the program units, but they *must* be in the correct order and have the correct types. It was possible to refer to a common block in as many program units as needed.

Common blocks could be distinguished by being given names. In the example above, the common block could have been given the name *data1* by declaring it as:

```
common /data1/  inum, x, array1
```

Note the following rules that applied for common blocks:

- A program could use as many named common blocks as needed, but the names had to be unique - they could not be the same as the names of subprograms and they could not match the names of variables in any subprogram that uses the common block.
- A variable could appear in only one common block.
- If a subprogram used a common block, none of the variables in the common block could appear in the argument list for the subprogram.

15.4 'Include' files

A common and useful extension to Fortran 77 allowed the use of **include** files. These could contain e.g. declarations of constants and common blocks so that these declarations need not be repeated in numerous subprograms.

A program could make use of the **include** file with an **include** statement, e.g:

```
include 'myincludefile'
```

which essentially inserted the text of the **include** file into the program. Like **common blocks**, include files have been superseded by **modules**.

15.5 Standard F77 DO loops

The only form of **do** loop that was included in standard Fortran 77 is shown in the following program fragment:

```
do 100 index=istart, iend, incr
   statement 1
   statement 2
   statement 3
   etc
100 continue
```

This **do** loop is contained between two statements: the **do** statement and a **continue** statement. (In this example, the loop uses statement label *100*, but it could have another number.) All the statements down to and including the line containing the statement label would be executed as in a **do…end do** loop. The **continue** statement is a Fortran statement which can be placed anywhere within a program but causes no action to be taken. It simply marks the end of the loop in this example.

16. Further information

The following books on Fortran 90 are available from the University Library. Those marked with an asterisk (*) are also available from the ITS Helpdesk:

Fortran 90/95 for scientists and engineers

by S.J. Chapman, published by McGraw-Hill (1998)

Fortran 90 programming

by T.M.R. Ellis, I.R. Philips and T.M. Lahey, published by Addison-Wesley (1994)

*Fortran 90/95 explained **

by M. Metcalf and J. Reid, published by Oxford University Press (1999)

Fortran 90 and engineering computation

by W. Schick and G. Silverman, published by Wiley (1995)

*Numerical Recipes in Fortran 90 **

by W. H. Press et al., published by Cambridge University Press (1996)

The documentation for the Portland compilers can be found at:

http:www.dur.ac.uk/portland/

Extensive documentation on the Sun compilers is available at:

http://docs.sun.com/

A web-based course on Fortran 90, developed at the University of Liverpool, is available on the ITS web pages at:

http://www.dur.ac.uk/its/local/fortran90/HTMLF90CourseSlides.html

For a summary of new features in Fortran 95 see:
http://www.kcl.ac.uk/kis/support/cit/fortran/f90home.html#1.2