PH25520
Experimental Physics
Introduction to Fortran 90

Daniel Brown
UNIVERSITY OF WALES ABERYSTWYTH

1 Introduction

1.1 Resources

While this document is self-contained, those interested in a Fortran 90 textbook should consult:

T.M. Ellis, I.R. Philips and T.M. Lahey, Fortran 90 Programming, Addison-Wesley.

There are copies in the library, although it is a text well worth purchasing if you envisage using Fortran 90 a lot in the future (e.g., for a numerical project or a fairly computational PhD).

In addition, you may want to download a Fortran 90 compiler for your own computer/laptop. Free and commercial Fortran compilers for windows and Linux can be found at the following websites

`http://www.fortran.com/`
`http://www.g95.org/`

The `F` compiler is a cutdown version that should work fine, G95 is an open-source Fortran 95 compiler that should be okay as well. Fortran 95 compilers are backwards compatible with Fortran 90.

Note, if you do download a compiler and write course material on your own computer, you should also test them on the course server - particularly programs that are to be assessed.

1.2 Telling a computer what to do

To get a computer to perform a specific task it must be given a sequence of unambiguous instructions or a program.

An everyday example is instructions on how to assemble a bedside cabinet. The instructions must be followed precisely and in the correct order:

1. insert the spigot into hole ‘A’,
2. apply glue along the edge of side panel,
3. press together side and top panels
4. attach toggle pin ‘B’ to gromit ‘C’
5. ... and so on

A programming language is a way to give a list of instructions to a computer so that the computer can carry out a task.

Programming languages must be:

- totally unambiguous (unlike natural languages such as English),
- expressive – it must be fairly easy to program common tasks,
- practical – it must be an easy language for the compiler to translate,
- simple to use.

All programming languages have a very precise syntax (or grammar). This ensures all syntactically correct programs have a single meaning.

1.3 High-level programming languages

In the early days of computing, all programs were written in machine code. These were just long strings of 0s and 1s in a binary form, such as,

```
010100011 010 000 010111
```

This had the disadvantage that it was specific to a particular type of computer and was almost totally incomprehensible to a human being.

The binary code became octal code (base 8) such as

```
243 2 0 27
```

This in turn evolved into assembler code which is mnemonic form of the machine code instructions, such as

```
LDA 2 X
```

which means *load CPU register 2 with the contents of memory location X*. These principals of machine code have survived more of less unchanged to this day.

This kind of programming was for the specialist programmers rather than the every day users.

In 1953, IBM decided that it would be beneficial if a more efficient and economical method for programming a computer existed, and by mid-1954 an initial specification had been written for a programming language. This language was called *IBM Mathematical FORmula TRANslation System* of FORTRAN.

FORTRAN introduced many important concepts, the main one being that the program was formulated in the users terms, and not those of the computer. This idea of using algebraic terms and a 'pidgin English' for other (non-mathematical) terms became known as a high-level language as the user did not need to know much about the specifics of the computer itself.

A computer can only understand its own machine code, so before a FORTRAN (or other high-level language) program can be run, it must be translated (or compiled) into the specific machine code for that computer. A special program (called a compiler) must be used to translate a FORTRAN program from the high-level code to the low-level machine code.

Nowadays, there are many high-level languages such as Fortran 90, FORTRAN 77, C, C++, COBOL, BASIC and JAVA. FORTRAN itself has changed dramatically over the years from Fortran I and II, to the first standardised version Fortran IV in 1966. And more recently FORTRAN 77 (in 1977), Fortran 90 (in 1990) and even Fortran 95.

1.4 An example problem and program

Consider the problem of how to convert from °F (Fahrenheit) to °C (Celsius). We can use the following formula:

$$C = \frac{5}{9}(F - 32)$$

To convert from °C to K (Kelvin) we add 273.

The program would accept a Fahrenheit temperature as input and produce the Celsius and Kelvin equivalent as output.

In Fortran 90, we might code this up as

```
PROGRAM temp_conversion
! this program take an input in Fahrenheit and converts it to
```

```
! both Celsius and Kelvin - outputting both results to the
! screen

! variable declarations
  IMPLICIT NONE
  INTEGER :: deg_F, deg_C, K

! read in Fahrenheit data
  PRINT*, "Please type in the temp in F"
  READ*, deg_F

! convert to Celsius and output
  deg_C = 5*(deg_F-32)/9
  PRINT*, "This is equal to", deg_C, "C"

! convert to Kelvin and output
  K = deg_C + 273
  PRINT*, "and", K, "K"

END PROGRAM temp_conversion
```

This program, called `temp.f90`, can be compiled an run in an X-terminal (on a UNIX system) as follows:

```
f90 temp.f90
a.out
```

1.5 Analysis of program

The code is delimited by the `PROGRAM` and `END PROGRAM` statements which indicate the main program code (there are other delimiters such as `FUNCTION`, `SUBROUTINE` and `MODULE`, these will be looked at later in the course). Between these there are two distinct areas of the program.

1. The specification part
   - specifies the named memory locations (variables) for use in the program
   - specifies the type of the variable
   - there are other specifications that will be described later in the course.
2. The execution part
   - reads in the data
   - calculates the temperature in °C and K
   - outputs (in this case print to screen) the results.

The `PROGRAM` statement has a program name attached to it. There a certain rules governing legal program names. These rules apply to all Fortran 90 user defined names (such as variable names and so on). These rules are

- It must begin with a letter - either upper or lower case
- It may only contain the letters A-Z and a-z, the digits 0-9 and the underscore character _

- It has a maximum length of 31 characters

The lines that begin with the character `!` are comment lines. Anything after the `!` is ignored when the program is compiled and is purely for the benefit of the programmer (or someone else who later reads through the program).

You should write comment lines to document your code and explain what is going on. Use as many comment lines as you need (possibly extra if you plan on passing your code on to someone else), but try to avoid needless comments where the code says it all, e.g.,

```
! set a equal to 1
a=1
```

In this case, the comment doesn't tell you anything extra than the code itself.

A more useful comment might be

```
! a is the radius of the circle
a=1
```

as it tells you what `a` represents. Comments can appear on the same line as some code, e.g.,

```
a=1          ! the radius of the circle
```

Though this should be used sparingly and only on short lines of code.

Documentation might be achieved more efficiently with self-documenting code using an appropriately named variable, e.g.,

```
rad_circ=1
```

It is also a good idea to put a brief description of what the program is supposed to do at the beginning of the code.

1.5.1 The specification part

The first statement in this part is

```
IMPLICIT NONE
```

This statement is used to inhibit an undesirable feature carried over from previous versions of Fortran. Using this statement means that you must declare all of the variables you use at the beginning of the program. You should always use this statement, and it is placed before all variable declarations.

```
INTEGER :: deg_F, deg_C, K
```

This statement is used to declare variables that you wish to use in your program and to tell the computer what type of variables they are.

In this case we are declaring three variable (`deg_F`, `deg_C` and `K`) that take integer values (whole numbers) only.

Some other variable types are

- `REAL` – real numbers such as 3.1415 and 5.213×10^{-4}
- `LOGICAL` – takes the values `.TRUE.` or `.FALSE.`
- `CHARACTER` – contains a single alphanumeric character, e.g., 'a'

- `CHARACTER(LEN=12)` – contains a string of 12 alphanumeric characters

1.5.2 The execution part

This is the part of the program that does the actual work. The key lines are as follows:

```
PRINT*, "Please type in the temp in F"
```

This writes the string (message) to the screen

```
READ*, deg_F
```

This reads a value from the keyboard and assigns it to the variable `deg_F`.

```
deg_C = 5*(deg_F-32)/9
```

The expression on the right hand side is evaluated using the inputted value of `deg_F` and is assigned to the variable `deg_C`. The expression performs the conversion from Fahrenheit to Celsius. The following operators are used

`+` The addition operator

`-` The subtraction operator

`*` The multiplication operator

`/` The division operator

`=` The assignment operator

The brackets, `()`, may be used to help specify the order that the calculation is performed.

```
PRINT*, "This is equal to", deg_C, "C"
```

This displays a sequence of outputs on the screen. First the string "This is equal to" followed by the value of the variable `deg_C`. This is then followed by the final string "C".

1.6 Programming conventions

Fortran is not case sensitive, so `K` is the same as `k` and `integer` is the same as `INTEGER`. This document will use the convention that everything that is part of the Fortran language (`INTEGER`, `PRINT*`, etc) will be typed in capitals letter. Everything that is defined by the user (such as variables and program names) will be typed in lower-case letters with the occasional capitol letter if it makes sense (such as the `C` in `deg_C`).

1.7 How to write a computer program

There are four main steps to writing a computer program. These are:

1. Specify the problem – decide (write down) what it is the program is supposed to achieve
2. Analyse the problem and break it down to a series of steps towards the solution

3. Write the Fortran 90 code

4. Compile and run the program (i.e., testing)

The first two steps may be best done using pen and paper. Especially for longer programs, it is useful (necessary) to have some kind of plan and a good idea of what steps are required before starting to code.

It may be necessary to repeat steps 3 and 4 many times before the program runs correctly. It is very rare that a code works first time.

The testing phase is very important. In larger projects it is usually possible to break the project up into many smaller chunks that can be coded and tested separately.

1.8 Compiling and running a program

The program can be created using a text editor such as notepad or emacs. Suppose you have created a Fortran 90 program called `myprog.f90`, this can be compiled at the UNIX prompt by typing

```
f90 myprog.f90
```

This will create an executable file (a machine code file) called `a.out` which can be run by typing

```
./a.out
```

When you create your executable, you can specify a different name by typing

```
f90 -o myprog.out myprog.f90
```

Which can be run with

```
./myprog.out
```

1.9 Free- and fixed-format Fortran

The original Fortran language used a so-called fixed format where the first 5 columns were used for labels, column 6 for a continuation character and columns 7-72 for code. This has much to do with early programming being performed on punch-cards where the formatting of the code was important.

Fortran 90 introduced a free-format as it had little need for the strict format that existed in previous versions of Fortran. Fixed-format code is still acceptable and files with fixed-format Fortran will end with the extension `.f`.

This course will only look at free-format code. The accepted extension for this is `.f90`, and all of your Fortran files should use this extension. Some compilers use this information to anticipate what type of Fortran is about to be compiled.

Free-format Fortran has only a few formatting rules, these are:

- Blank characters (spaces) are important and must be used to separate names, constants or statement labels from other names, constants, statement labels or Fortran keywords
- Comment lines are identified by having an exclamation mark (`!`) as the first non-blank character
- Any characters following an exclamation mark (unless this is part of a character string) forms the trailing comment

- A line may contain more than one statement, in which case each statement must be separated by a semi-colon (`;`)
- A line may contain a maximum on 132 characters
- A trailing ampersand (`&`) indicates that a statement is continued on the next line. If it occurs in a string context then the first non-blank character of the next line must also be an ampersand, and the string continues immediately after the ampersand
- A statement may have a maximum of 39 continuation lines
- A statement label, if required, consists of up to five consecutive digits representing a number in the range 0-99999. This precedes the statement and is separated from it by at least one space.

Statement labels are rarely needed in Fortran 90 and are included more for historical reasons. We will barely touch upon statement labels, if at all.

The rules above discuss a maximum length of a line and continuation lines. If you have a line that is longer than the maximum, then you need some mechanism to split it over two lines, this is where the continuation character, `&`, comes in. When a line ends with the continuation character then the Fortran compiler knows that the Fortran statement is not yet finished. For example, the following two statements do the same thing;

```
mysum = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
mysum = 1 + 2 + 3 + 4 + 5 &
          + 6 + 7 + 8 + 9
```

In the second statement, the sum is continued on the next line (also, note the use of spaces to aid visual formatting).

You need to be slightly careful if you want to split a text string, for example;

```
print*, "The rain in Spain falls mainly on the plain."
print*, "The rain in Spain &
        &falls mainly on the plain."
```

In this case an additional continuation character is required at the beginning of the continuation line. The program will try and print valid spaces, so the continuation symbol indicates where the string resumes.

This additional continuation character at the beginning of the continuation line works for normal lines as well. So for our sum example, the following statements do the same thing

```
mysum = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
mysum = 1 + 2 + 3 + 4 + 5 &
          + 6 + 7 + 8 + 9
mysum = 1 + 2 + 3 + 4 + 5 &
      &   + 6 + 7 + 8 + 9
```

The use of continuation characters at the beginning of a continuation line gives more legible code (when combined with sensible spacing), this option is recommended when using continuation lines.

Note, the one place where continuation lines do not work is in comments, so

```
! This is going to be a long comment &
& which is spread on two lines
```

will cause a compiler error. The correct way to do this is

```
! This is going to be a long comment
```

```
! which is spread on two lines
```

Additional note, although the maximum length of a statement line is 132 characters, most printers and text editors default to around 80 characters per line. It makes for more legible code if you keep your lines below this limit. Furthermore, When splitting lines, you don't have to split them at character 80, if there is a natural break in the statement, use that instead. For example,

```
mycalc = (x-a)/SQRT((x-a)**2 + (y-b)**2) + (y-b)/SQRT((x- &
       & a)**2 + (y-b)**2)
```

gives an ugly break, where

```
mycalc =   (x-a)/SQRT((x-a)**2 + (y-b)**2) &
       & + (y-b)/SQRT((x-a)**2 + (y-b)**2)
```

is a more natural split and makes the equation easier to read.

2 Introduction to UNIX

2.1 What is UNIX

UNIX is a command-line operating system (rather than the point-and-click Microsoft operating systems) that is intended for a more 'expert' user. There are many different flavours of UNIX such as Linux, DEC UNIX, SUN UNIX and many more.

UNIX is used extensively for serious applications. Scientific programming is often done under UNIX. Other applications are central file servers, web servers and supercomputers. Much of the cgi animation that you see in TV and films is done on supercomputers running some type of UNIX.

In computational physics, the aim is to program and perform complex simulations of some physical system (such as fluid flow or plasma dynamics). These simulations are often designed to to be run on parallel computers where there may be 100s of computers (nodes) each performing a bit of the simulation and passing on the necessary information from it's own bit for other nodes to continue with the simulation. Almost all parallel computers run some flavour of UNIX.

If we are interested in doing computational physics (which is why we are learning Fortran), then we are interested in learning how to use UNIX.

2.2 Logging on

For this course, we will be using `fortran.dph`, a UNIX server set up especially for this course. To log in from one of the workstations in the honours laboratory do the following;

First, we must start the X-server on the windows machine, in the `All Programs` list find the `Cygwin` menu, from this select the `Start X-Server` option.

After a few moments, a terminal window running on the local desktop will appear. This is actually a UNIX shell emulator. You will now need to log in to the fortran server from this terminal window. To do this type

```
ssh -X -Y fortran.dph
```

This command is the secure shell command for connecting into other machines. The `-X` flag tells it that you want to enable X-forwarding (which you do in this case), the `-Y` flag enables trusted X-forwarding, and the following argument tells it which server to connect to.

If you want extra/new terminal windows after you have started the X-server, from the `All Programs` list find the `Cygwin-X` menu, and from this select the `xterm` option.

When you are logged in, the prompt in your terminal window should change to something involving your username and the server name, for example

```
dob@fortran:~>
```

In this document, the prompt will be written with any command whenever you are asked to type something in under UNIX. You should not rewrite the prompt, just the command that follows it.

For example, make a new directory for the Fortran 90 source code by typing:

```
dob@fortran:~> mkdir Fort90
```

It is suggested that when you name new directories, the first letter should be capitalised to differentiate directories from regular files in directory lists.

You can change to the new directory by typing

```
dob@fortran:~> cd Fort90
```

2.3 Creating and editing files

The command `cat` ('catalogue') displays the contents of a file on the screen. You can use this to create a new file (called *`myfile`*) as follows:

```
dob@fortran:~> cat > myfile
Type first line
Type second line
.........
Ctrl-d
```

The final line means hold down the control key (labelled 'Ctrl') and then press 'd'.

You can then display the contents of this file by typing:

```
dob@fortran:~> cat myfile
```

Where the $>$ symbol is missing.

By using a text editor, this file can be edited and changed. There are many text editors in UNIX, most of which are rather difficult to use. The most widely available, simple to use editor under UNIX is called emacs. To use this type:

```
dob@fortran:~> emacs myfile
```

and use the intuitive interface. If you supply a new file name, a new file with that name will be created.

You will notice that when you do this, you are unable to enter any new commands in the UNIX terminal. If you type

```
dob@fortran:~> emacs myfile &
```

then emacs will be run as a background process and you will be able to enter commands at the UNIX terminal.

Important note: the files on Fortran.dph are not backed up. When working on important files, you will have to back them up yourselves. You can transfer files to your workstation by using the WinSCP tool. There should be a shortcut on the desktop with the WinSCP icon and Fortran written underneath. This will connect you to the Fortran server and you can then copy files to your m: drive.

Similarly, you can copy files you have created elsewhere to the Fortran server using this tool.

2.4 Other UNIX commands

2.4.1 Displaying a file on screen

We have already seen that the `cat` command will display the content of a file on the screen. However, if the file is long, the beginning scrolls of the top of the terminal.

A more useful command is `more`. This will display the file one page at a time.

```
dob@fortran:~> more myfile
```

Pressing the space bar or a carriage return will advance the text by one page. typing `100f` will advance

by 100 lines and typing `100b` will go back 100 lines. Typing */string* will move forward to the next occurrence of that string.

2.4.2 Deleting files and directories

The command for deleting files is `rm`. To remove a specific file type

```
dob@fortran:~> rm myfile
```

Typing

```
dob@fortran:~> rm *
```

will remove every file in the current directory, `*` is a wild card and will match any string. This can be dangerous, as you may delete files you want to keep (there is no undelete or recovery from a trashcan in UNIX), a safer way to do this is

```
dob@fortran:~> rm -i *
```

This sends a y/n query to the screen for every file that it matches.

Another wild card is `?` which matches exactly one single character, so typing

```
dob@fortran:~> rm file??
```

will delete files that have names like `file01`, `file02`, `fileab`, , `file.f`, etc.

To remove a subdirectory, you must first delete all the files contained within the directory, then you type

```
dob@fortran:~> rmdir subdir
```

2.4.3 Directory listings

To obtain a listing of which files are contained withing a directory, type

```
dob@fortran:~> ls
```

This will give a list of all the user files and subdirectories contained in the current working directory.

To see the contents of a subdirectory, type

```
dob@fortran:~> ls subdir
```

You may use wild card characters (`*` and `?`) to list a selection of files, for example

```
dob@fortran:~> ls *.f90
```

will list all the Fortran 90 (or at least all the files ending `.f90`) in the current directory.

To get more details about the files (such as last date modified, size and ownership of file) you can use the `-l` (long) option,

```
dob@fortran:~> ls -l
dob@fortran:~> ls -l subdir
dob@fortran:~> ls -l *.f90
dob@fortran:~> ls -l subdir/*.f90
```

The last example displays more detail of all Fortran 90 files contained within the subdirectory *subdir*. The `/` is a divided between the subdirectory and the file selection. You may have many such dividers (as long as those subdirectories exist, so `Fort90/Sect2/Ex4/*.f90` would refer to all Fortran 90 files

within the subdirectory `Ex4` which is contained within the subdirectory `Sect2` which in turn is in the directory `Fort90`.

Other useful options to the `ls` are

```
dob@fortran:~> ls -a
dob@fortran:~> ls -R
```

The `-a` (all) option lists all files including hidden system files, and the `-R` (Recurse) lists all files in the current directory and all of its subdirectories.

You may usually be able to use multiple option, e.g.,

```
dob@fortran:~> ls -la
dob@fortran:~> ls -l -a
```

will both list all the files and system files within a directory and supply the file details.

2.4.4 Copying and renaming files

Often you may wish to make a copy of a file (perhaps you want to use an existing Fortran program as a template for another). Copying can be done as follows

```
dob@fortran:~> cp file1 file2
```

You may wish to copy a file from one subdirectory to another, e.g.,

```
dob@fortran:~> cp subdir1/file1 subdir2/file2
```

You may wish to copy a file from one directory to another, without changing the name, the following two commands both do this

```
dob@fortran:~> cp subdir1/file1 subdir2/file1
dob@fortran:~> cp subdir1/file1 subdir2/
```

The second example uses a shorthand, if you supply a directory name as a destination without a filename, UNIX will use the existing filename.

You can use this to copy many files to a new directory using wild cards. E.g.,

```
dob@fortran:~> cp subdir1/*.f90 subdir2/
```

will copy all Fortran 90 files in *`subdir1`* to *`subdir2`*.

Sometimes you will want to rename or move a file without making the extra copy. This can be done using

```
dob@fortran:~> mv file1 file2
```

which renames *`file1`* to *`file2`*. All of the things above that worked for `cp` will also work for `mv`, e.g.,

```
dob@fortran:~> mv subdir1/*.f90 subdir2/
```

will move all of the Fortran 90 files in *`subdir1`* to *`subdir2`* (leaving *`subdir1`* empty of Fortran 90 files).

2.4.5 Working within directories

Once you have a directory structure, you need to be able to move about within that structure. You can change the current working directory by

```
dob@fortran:~> cd subdir
```

This moves into a subdirectory of the current directory, called *subdir*.

```
dob@fortran:~> cd ..
```

Moves you back up a directory, so if you're in *subdir* and enter this command, you will be moved out of *subdir* and into the directory that contains it.

The `..` can be used in conjunction with other subdirectories, so

```
dob@fortran:~> cd ../otherdir/somedir/
```

Will move you up a directory then down the *otherdir/somedir/* branch.

If you just type

```
dob@fortran:~> cd
```

by itself, you will be taken back to your home directory.

If you're not sure where you are in your directory tree, type

```
dob@fortran:~> pwd
```

This will display the complete directory branch that you are in. E.g.,

```
dob@fortran:~> pwd
/mntce/staff2/base/d/dob/Fort90/Sect2/Ex2
```

You will notice that your home directory is several directories down the directory tree.

2.4.6 Printing a file

Most printers attached to a UNIX machine expect postscript pages (postscript is a computer language that tells printers how to print things). The basic command for printing a postscript file is

```
dob@fortran:~> lpr file.ps
```

However, this will send your output to the default printer which may, in general, be somewhere that you will not be able to find. For the Fortran server though, the default printer is the one in the honours laboratory.

Otherwise, you need to find out the name of the printer queue for a printer which you know the location of and include that information as follows

```
dob@fortran:~> lpr -Plocalprinter file.ps
```

where *localprinter* is the name of the local print queue. A list of suitable printers should be available in each terminal room.

This command often works for text-files as well, however, a safer way to send your text-files (such as your `.f90` programs) to the printer is as follows

```
dob@fortran:~> a2ps -Plocalprinter file.txt
```

Many applications, such as emacs, will have a print option so you can print directly from that.

If you wish to query the local printer (see what jobs have been submitted and where you are in the queue), type

```
dob@fortran:~> lpq -Plocalprinter
```

To remove a job (only one sent by you) type

```
dob@fortran:~> lprm -Plocalprinter jobno
```

where *jobno* can be obtained using the `lpq` command.

2.5 Further information

There is a large amount of information about UNIX available. If you are after detailed information about a specific command, you can usually consult the man page by typing

```
dob@fortran:~> man command
```

So to find information on the `ls` command you would type

```
dob@fortran:~> man ls
```

If you are after information about something but are unsure of the specific command, you can do a keyword search

```
dob@fortran:~> man -k keyword
```

Note that man pages are usually intended as a quick lookup rather than a tutorial and are not always written in a way suitable for a UNIX novice.

There are other sources online. Two useful places to start are:

College help page
http://www.inf.aber.ac.uk/publications/documentation/h1.asp

Departmental web page (information for current students)
http://www.aber.ac.uk/physics/unix_cmds.shtml

Ex 2.1 Log onto central and create yourself a new directory (perhaps called `Fort90`).

Change into this directory and create a new subdirectory for Fortran programs from this section (perhaps called `Sect2`). Change into this new subdirectory.

Ex 2.2 Using a text editor, type out the Fortran program from section 1 (included below). Save it as Fortran 90 file (something like `temp_fk.f90`) in the new subdirectory created above.

```
PROGRAM temp_conversion
! this program take an input in Fahrenheit and converts it to
! both Celsius and Kelvin - outputting both results to the
! screen

! variable declarations
  IMPLICIT NONE
  INTEGER :: deg_F, deg_C, K

! read in Fahrenheit data
  PRINT*, "Please type in the temp in F"
```

```
  READ*, deg_F

! convert to Celsius and output
  deg_C = 5*(deg_F-32)/9
  PRINT*, "This is equal to", deg_C, "C"

! convert to Kelvin and output
  K = deg_C + 273
  PRINT*, "and", K, "K"

END PROGRAM temp_conversion
```

Compile and run this program as follows

```
dob@fortran:~> f90 -o temp_fk temp_fk.f90
dob@fortran:~> ./temp_fk
```

Try running this program several times for different input values for `deg_F`. Do the answers agree with conversions worked out by hand/on a calculator.

Ex 2.3 In this question you will write the reverse program that takes input in Kelvin and converts to Celsius and Fahrenheit.

First write down (on a piece of paper) the equation to convert Kelvin to Celsius and the equation to convert Celsius to Fahrenheit.

Now make a plan (again on paper) of the steps you need to perform in the program (put in as much detail as you need to).

Now write the Fortran program (pick a new filename that bears some relation to what the code does), compiling, testing and modifying as required.

Try using your original program `temp_fk` to convert from Farenheit to Kelvin, then using that result in your new program to convert back to Farenheit. Do you always get exactly the same result you started with? If not, why not?

3 Data Types and Handling

3.1 Data Types

There are four basic types of data in Fortran (though these have subtypes). Two of these are the fundamental number types of integers (whole numbers) and reals (numbers with a fractional component). There is also character data and logical type data. We will look at logical data types later in the course.

3.1.1 Integer Type

Integers are whole numbers without any decimal/fractional component. When you store an integer in computer memory, there are limits on it's size.

As an example, consider a milometer (or odometer) on a car that has eight dials that each go from 0 to 9. Initially, the odometer has the reading;

0	0	0	0	0	0	0	0

After driving two miles the odometer reads

0	0	0	0	0	0	0	2

And after driving a lot more miles, the odometer reads

9	9	9	9	9	9	9	9

If the car drives another mile, then the odometer is going to turn over and all the dials will be back to the beginning again

0	0	0	0	0	0	0	0

So we're limited to the maximum reading on the odometer by the number of dials.

A computers memory works in a similar way (except computer memory is usually binary rather than decimal) and the maximum number that can be stored depends on the amount of memory that is allocated to the storage. If one byte (which is eight bits) is allocated then the maximum number is $2^8 - 1 = 255$.

This is fine, but it does not take into account negative numbers. Consider the odometer once more, but use the convention that 1 to 49 999 999 are considered to be positive numbers and that 50 000 000 to 99 999 999 translate to the negative numbers -50 000 000 to -1. So the following values on our odometer would translate as follows;

5	0	0	0	0	0	0	0	Represents -50 000 000
5	0	0	0	0	0	0	1	Represents -49 999 999
9	9	9	9	9	9	9	9	Represents -1
0	0	0	0	0	0	0	0	Represents 0
0	0	0	0	0	0	0	1	Represents +1
4	9	9	9	9	9	9	9	Represents +49 999 999

In Fortran 90, when you specify that you want a number to be an integer, it will store the number using this notation (though with binary arithmetic rather than decimal). So the range of numbers that you can

have depends on the amount of memory allocated to storing the integer.

3.1.2 Real Type

Real numbers are those with a whole number (integer) part and a fractional (or decimal) part. Again, when you store a real number there are limits on its size.

One could use a similar method to store real number as was used for storing integers, but with some of the positions/dials being after the decimal point (for example, on a milometer the final dial may be 10ths of a mile). For general storage of real numbers, this is impractical as you would need a lot of memory to store large numbers, but lots of this would be wasted. Also, multiplying small numbers would cause errors.

Instead, it makes more sense to use scientific notation such as 0.5×10^6 or 0.72×10^{-3} and so on. So let us consider the odometer again. This time take the first two dials to be the exponent and the remaining 6 dials to be the fractional part after the decimal point. So

0	6	3	6	2	0	9	4

Represents $0.362\ 094 \times 10^6$

In order to get negative real numbers and negative exponents we use the same method as before. So for the exponents, 0 to 49 is positive and 50 to 99 are negative and map to -50 to -1. Similarly, the fractional part can be mapped so that 0.5 to 0.999999 becomes -0.5 to -0.000001 (this works a lot better in binary than decimal).

So the following readings translate as follows;

0	3	2	7	6	0	3	1

Represents $0.276\ 031 \times 10^3 = 276.031$

0	4	7	4	0	1	9	3

Represents $-0.259\ 807 \times 10^4 = 2\ 598.07$

9	7	2	7	6	0	3	1

Represents $0.276\ 031 \times 10^{-3} = 0.000\ 276\ 031$

9	6	7	4	0	1	9	3

Represents $-0.259\ 807 \times 10^{-4} = 0.000\ 025\ 980\ 7$

So with this notation, you can have real numbers between -0.5×10^{49} to -0.499999×10^{49} and the smallest number you can have is $0.000001 \times 10^{-50} = 10^{-56}$.

Of course, a real computer will store this using binary dials rather than decimal ones.

3.2 INTEGER and REAL Variables

Suppose we have an odometer that has the reading

0	4	3	6	4	0	7	9

This could be interpreted as the integer $4\ 364\ 079$ or as the real number $0.364\ 079 \times 10^4$. In order to interpret the reading we must also label the odometer accordingly. We would probably want to label the odometer so we knew what it referred to (e.g., no of people, miles travelled, speed, etc). Something like the following may be appropriate;

n			Integer				
0	0	1	5	4	2	3	8

which represents $n = 154\ 238$

x						Real	
0	6	4	7	2	2	6	3

which represents $x = 0.472\,263 \times 10^6$

So we now know how to interpret the odometer reading and to what it refers.

This is similar to what occurs when we define variables in Fortran 90. When a variable is initialised as an integer or real number (or one of the other types), the computer reserves the amount of memory required to store the value as well as memory to store the type of variable it is and what its label is (and perhaps other properties of the variable).

So to define an integer we would write

```
INTEGER :: i
```

This would allocate memory for an integer variable with the label `i`. We can define many variable with the same command, e.g.,

```
INTEGER :: i, j, k
```

Similarly, to define a real variable we would write

```
REAL :: x
REAL :: x, y, z
```

3.2.1 What are the minimum and maximum values I can store?

We have seen above that integer and real variables have minimum and maximum values that can be stored, which depends on the amount of memory that the computer allocates for storage. Unfortunately, there is no standard value for all computers, and different machines may allocate different amounts of memory.

However, many machines are 32 bit machines and these usually allocate 32 bits (or 4 bytes) to integer or real variables. For an integer variable this gives a range of $-2\,147\,483\,647$ to $2\,147\,483\,647$. For real variable, this gives a range of about -10^{38} to 10^{38} with 7-8 significant figures.

With the advent of 64 bit computers, this may well be different (though the range will likely be larger with better accuracy). Much older computers may give smaller ranges.

3.3 Arithmetic and Assignment

3.3.1 Assignment

Once we have defined our variables, we want to be able to assign values to them and use these values in arithmetic expressions. There are two ways to do this, the first is by the assignment operator (`=`) and the second is by the `READ` statement. We met both of these in our temperature conversion program in section 1.4.

The assignment operator works as follow. Fortran 90 evaluates the expression on the right hand side of the `=` operator and assigns it to the variable who's label appears on the left hand side. For example

```
INTEGER :: i, j, k

i = 4          ! assign the value 4 to i
j = i + 2      ! add 2 to the value of i and assign to j
k = j - i      ! subtract the value of i from j and assign to k
```

```
i = i + 1     ! add 1 to the current value of i and assign it
              ! back to i
```

In the last example, the variable `i` appears on both sides of the operator. So the old value of `i` is used to evaluate the right hand side and this new value is then stored under the variable `i`. This does not go back and change the values of `j` or `k`.

3.3.2 Arithmetic

In the previous example we saw that the `+` and `-` operators were used to represent addition and subtraction. The basic arithmetic operators are as follows;

Operator	Meaning
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation (or “raise to the power of”)

Using these operators we can construct arbitrary complex expressions. For example

```
a=b+c*d/e-f**g/h+i*j+k
```

For statements like this it is important to know in what order Fortran will evaluate the expression. For this situation, Fortran has an operator priority (which is basically the same as in mathematics) which is outlined in the following table.

Operator	Priority
`**`	high
`*` and `/`	medium
`+` and `-`	low

So Fortran will carry out all the high-priority operations first, followed by all the medium-priority operations and finally all the low-priority operations. If there are several operations of the same priority, then Fortran works from left to right.

So for the above expression, the order in which operations are done can be written as follows

```
temp1 = f**g
temp2 = c*d
temp3 = temp2/e
temp4 = temp1/h
temp5 = i*j
temp6 = b + temp3
temp7 = temp6 - temp4
temp8 = temp7 + temp5
a = temp8 + k
```

In practise, Fortran doesn’t actually assign to `temp?` variables, instead it uses special high-speed memory locations (called registers) to speed up the calculation.

Ex 3.1 Consider the calculation

```
a=b*c+d/e**f-g+h*i*j
```

Write a piece of Fortran 90 code that calculates this in (a) a single line; and (b) a series of lines with one operation per line in the order that Fortran would carry them out. Use `REAL` variables only in this code.

Put in some test values for `b-j` and output the result of your calculations for both cases to check that they both give the same answer.

3.3.3 Parentheses

There will be times when you want your calculation (or at least parts of it) to be carried out in a different order than described above. For example, if you were calculating the mean of two values you would calculate

$$\frac{a+b}{2}$$

where the addition is performed before the division. You could split this calculation into two parts, so

```
temp1=a+b
mean=temp1/2.0
```

However, it is more convenient to write this on one line using parentheses to indicate the order in which the calculation should be executed. E.g.,

```
mean=(a+b)/2.0
```

Parentheses change the order of the calculation in the same way that they do in mathematics. When in doubt about the order in which a calculation is being executed, add appropriate parentheses in order to clarify things (at the very least, this may make it clearer for anyone reading your code).

Ex 3.2 Modify the code that you wrote in exercise 3.1 to calculate

```
a=b*c+d/e**f-g+h*i*j
```

Try adding parentheses around different parts of the equation to change/not change the final result.

3.3.4 Mixed Modes and Integer Division

Ex 3.3 Consider the following Fortran code;

```
PROGRAM mixedmodes

  IMPLICIT NONE
  INTEGER :: i, j
  REAL :: x, y1, y2

  i=2
  j=3
  x=5.0

  y1=x*i/j
  y2=i/j*x

  PRINT*, 'y1=',y1
  PRINT*, 'y2=',y2
```

```
END PROGRAM mixedmodes
```

According to the rule governing the order of execution, the two formulae should give the same result. What happens when you type this in and run it?

Modify the code so that it also calculates `i/j` and output the result. Is this what you expected?

From exercise 3.3, you may have guessed that there is something not quite right about integer division. For the other operators (`+`, `-`, `*` and `**`), if they operate on two integers then they return an integer as an answer (so adding two integers returns another integer). From a programming perspective, you would want division to behave the same way. Clearly though, dividing one integer by another does not necessarily return another integer. In order to make integer division return an integer, Fortran truncates the result. So $2/3 = 0.6666$ which Fortran truncates to 0.

When both `INTEGER` and `REAL` variables are used in a single calculation then Fortran converts between the two as it deems necessary for each individual operation. So for our two calculations above

```
! first calculation
  y1=x*i/j
! performs calculation as follows
  temp1=x*i     ! converts i to a REAL and multiplies by x
                ! stores as REAL variable temp1
  y1=temp1/j    ! converts j to a REAL and divides temp1 by it
                ! stores as a REAL variable

! second calculation
  y2=i/j*x
  temp2=i/j     ! performs integer division and stores as an
                ! INTEGER variable temp2
  y2=temp2*x    ! converts temp2 to a REAL and multiplies by x
                ! stores as a REAL variable
```

So for each individual part of the calculation, if both variables are `INTEGER`s, then Fortran will perform integer arithmetic. If one is an `INTEGER` and the other is `REAL` then the integer will be converted to a real number before the calculation in done.

When the final result is stored, Fortran will convert the final answer to the type of the variable that it is being stored in. So if you try and store an integer result in a real variable, then it will be converted to a real number first. If you try to store a real result in an integer variable, then the real number will be truncated to an integer first.

Clearly this kind of mixed mode calculation and integer division should generally be avoided unless you are specifically after this kind of effect.

3.3.5 Converting Between `INTEGER` and `REAL` Variables

It is all very well saying that you should avoid mixed mode calculations, but there are often times when you need to use something stored as an `INTEGER` in a real calculation. In these cases, it is usually most sensible to use a conversion function. So to convert an `INTEGER` to a `REAL` you would use the following function;

```
x=REAL(i)
```

This converts the integer `i` into a real number and stores it in the real variable `x`.

There are four functions that Convert from `REAL` to `INTEGER`, each doing so in a slightly different way. These are;

Function	Effect
`AINT(x)`	Truncates the real leaving the whole part
`CEILING(x)`	Rounds the real number down to the next integer
`FLOOR(x)`	Rounds the real number up to the next integer
`NINT(x)`	Rounds the real number to the nearest integer

These functions can be included within a larger line of calculation, e.g.,

```
a = REAL(i+j)/x + FLOOR(y)/CEILING(y)
```

Ex 3.4 Modify the code in exercise 3.3 so that the integers `i` and `j` are converted to real variables within the calculations for `y1` and `y2`. Do the results for each calculation now agree?

Ex 3.5 Write a Fortran code to calculate and display the results of `AINT(x)`, `CEILING(x)`, `FLOOR(x)` and `NINT(x)` for each of the following cases;

(a) `x=2.34`

(b) `x=4.61`

(c) `x=1.5`

(d) `x=-1.13`

(e) `x=-3.72`

(f) `x=-0.5`

Test with some different values until you appreciate the difference between the four functions.

3.3.6 Unary Operators

As can be seen in exercise 3.5, the `-` operator can be used as a unary operator (i.e., it acts only on a single value). The `+` operator can also be used as a unary operator (although this is rarely needed for obvious reasons).

In both cases, the operator acts as you would expect. For example;

```
x=-3.1
i=+2
```

3.3.7 Spacing Things Out

Fortran will ignore any spaces in arithmetic expressions. This means that you can add spaces into such expressions without any effect. While this will not change how the code runs, it can make expressions (particularly longer ones with several terms) easier to read. For example;

```
a=b*c+d/e**f-g+h*i*j
a = b*c + d/(e**f) - g + h*i*j
```

In this case, the expression has not been changed, but the addition of spaces to separate the different terms and the addition of a pair of brackets makes the expression easier for the programmer to read.

It is up to you as a programmer exactly how to format expressions, but it is recommended that some visual formatting of your code is done as this will aid you when you debug your code or look back at it later. It will also assist other people who look at your code.

3.4 Literal Constants

Literal constants are just numbers. However, in Fortran, there are different ways of writing numbers depending on what you want.

If you want to write an integer, you just type the number without a decimal point. So the following are `INTEGER` constants;

```
i = 246
i = 20000
i = -7
i = 0
```

Real numbers are indicated by a decimal point (even if the fractional part is `0`). The following are `REAL` constants;

```
x = 3.12
x = 4000.0
x = -14.769
x = 0.0
```

There is another way to write `REAL` numbers which is particularly useful for very large or very small numbers, this is called *exponential form*. This takes the form

*m*E*e*

Where *m* is called the mantissa and *e* is the exponent. The mantissa may be written with or without a decimal point, and the exponent must take the form of an integer. So the number $1\ 000\ 000 = 1^6$ can be written as

```
x = 1.0E6     ! with a decimal point
x = 1E6       ! without a decimal point, but is still a REAL
```

The following are more examples of `REAL` constants in exponential form;

```
x = 2.64E27
x = 0.75E-5
x = -3E-12
x = 256E6
```

3.5 Input and Output of Numerical Data

We have come across input and output in the form of the `PRINT` and `READ` statements in the temperature conversion program in section 1.4. We shall now look at them again in more detail with the knowledge we now know about real and integer variables.

In the form we shall currently be using them, they have almost identical syntax;

```
READ *, var1, var2, ...
PRINT *, item1, item2, ...
```

The main difference is that the list following a `READ` statement may only contain variables, where the

list following a PRINT statement may also contain literal constants and expressions. The list following the READ and PRINT statements are often referred to as the *input list* and *output list* respectively. The asterisk following the READ and PRINT statements indicates that *list-directed formatting* is to take place. There are other types of input and output formatting, but they are beyond the scope of this course.

The list-directed READ statement will take it's input from a processor defined input. For most computers this will be the keyboard. Similarly, the list-directed PRINT statement will send it's output to a processor defined input, which is usually a terminal on the computer screen. We will see how to read from and print to files later on in the course.

3.5.1 The READ Statement

The statement

```
READ *, real_var1, real_var2, int_var
```

will read a list of three values from the input device (e.g., keyboard). it will then store these value in the variables real_var1, real_var2 and int_var which are two real variables and an integer respectively. If a number to be stored in a real variable does not have a decimal point (i.e., is an integer), then it will be converted to a real. If a number to be stored as an integer contains a decimal point, then this will cause an error.

As the term list-directed implies, the inputs are read in as a list. This means that we need some way to separate different terms in the list when we type them in. For this we use a *value separator* between values of the list. A value separator can be a comma, a space, a slash (/) or an end of line. Any of these can be preceded by any number of additional spaces.

If there are two consecutive commas, then this is interpreted as a *null value*. This results in the value of the variable being unchanged. A common error is to assume that a null value will set a variable to 0.

Finally, if the terminating character is a slash, then no more data items are read and any remaining items in the input list are give null values (i.e., are unchanged).

Ex 3.6 Type in the following program.

```
PROGRAM input_example

  IMPLICIT NONE
  INTEGER :: int1, int2, int3
  REAL :: real1, real2, real3

! initialise all variables
  int1 = 0
  int2 = 0
  int3 = 0
  real1 = 0.0
  real2 = 0.0
  real3 = 0.0

! read in data
  READ *, int1, real1, int2, real2, int3, real3

! print new values
  PRINT *, int1, real1, int2, real2, int3, real3
```

```
END PROGRAM input_example
```

Compile and run the program and enter the following values;

(a) `1,2.0,3,4.0,5,6.0`

(b) `1 2.0 3 4.0 5 6.0`

(c)
```
1 2.0
  3 4.0
  5 6.0
```

(d) `1,,,4.0,,6.0`

(e) `1, , , 4.0, , 6.0`

(f) `1 2.0, 3 /`

(g) `/`

Now change the initial values of the variables in the code and retry the above tests.

Try more test values until you are happy that you understand what is going on.

3.5.2 The `PRINT` Statement

We have seen the `PRINT` statement in action several times so far. The statement

```
PRINT *, var1, var2, var3
```

will display the values `var1`, `var2` and `var3` to the default output (usually the screen). The format of the output is entirely computer dependent, but is normally adequate for simple programs and testing. More advanced formatting is possible, but beyond the scope of this course.

As has been mentioned already, the output list of the `PRINT` statement can contain variables, literal constants and expressions. So the statement

```
PRINT *, 'The area of the circle of radius ', r, ' is ', 3.14*r**2
```

has a list containing four items. These are

1. The character string (literal constant): `The area of the circle of radius`
2. The variable: `r`
3. The character string: `is`
4. The expression: `3.14*r**2`

When this is run, Fortran will substitute the value of `r` and the value of the evaluated expression `3.14*r**2` into the output list.

Ex 3.7 Write a program that reads in the value of the radius of a circle (don't forget to prompt the user for input) and prints out (a) the radius, (b) the diameter, (c) the circumference, and (d) the area of the circle. You may only use a single variable for the radius, all other quantities must be calculated as expressions on the `PRINT` line. You may use π to two decimal places ($\pi = 3.14$).

3.6 Intrinsic Functions

Intrinsic functions are mathematical (or other) functions that are built into the Fortran 90 language. We have seen some implicit functions in section 3.3.5 for converting between variable types, such as the `REAL` function. There are many intrinsic functions in Fortran, a selection of the more useful mathematical ones can be found in table 1.

Function	**Input**	**Output**	**Purpose**
`ABS(A)`	integer or real	as input	returns the absolute value of `A`
`ACOS(x)`	real, $\|x\| \leq 1$	$0 \leq$real$\leq \pi$	returns the inverse cosine (or arccosine) of `x`
`ASIN(x)`	real, $\|x\| \leq 1$	$-\pi/2 \leq$real$\leq \pi/2$	returns the inverse sine (or arcsine) of `x`
`ATAN(x)`	real	$-\pi/2 \leq$real$\leq \pi/2$	returns the inverse tangent (or arctangent) of `x`
`ATAN2(y,x)`	both real	$-\pi \leq$real$\leq \pi$	returns the inverse tangent (or arctangent) of `y/x`, both x and y cannot be 0
`COS(x)`	real	real	returns the cosine of `x`
`COSH(x)`	real	real	returns the hyperbolic cosine of `x`
`EXP(x)`	real	real	returns e raised to the power `x`
`FRACTION(x)`	real	real	returns the fractional part of `x`
`LOG(x)`	real	real	returns the natural logarithm of `x`
`LOG10(x)`	real	real	returns the logarithm of `x` to base 10
`MAX(A1,A2,...)`	all integer or all real	as input	returns the maximum value of `A1`, `A2`, ...
`MIN(A1,A2,...)`	all integer or all real	as input	returns the minimum value of `A1`, `A2`, ...
`MOD(A,P)`	both integer or both real	as input	returns `A` modulo `P` calculated by `A - P*AINT(A/P)`
`MODULO(A,P)`	both integer or both real	as input	returns `A` modulo `P` calculated by `A - P*FLOOR(A/P)`
`SIGN(A,B)`	both integer or both real	as input	returns the absolute value of `A` set to the same sign as `B`
`SIN(x)`	real	real	returns the sine of `x`
`SINH(x)`	real	real	returns the hyperbolic sine of `x`
`SQRT(x)`	real	real	returns the square root of `x`
`TAN(x)`	real	real	returns the tangent of `x`
`TANH(x)`	real	real	returns the hyperbolic tangent of `x`

Table 1: Table of selected mathematical intrinsic functions.

These functions can be included directly into expressions or assigned to variables in the same way that the `REAL` function can. The following are all examples of how intrinsic functions may be used.

```
y = SQRT(x)
y = SQRT(2.0)
z = (a1*COS(k1*x) + a2*SIN(k1*x))*EXP(k2*t)
PRINT *,'The logarithm of ', x, ' is ', LOG(x)
```

Ex 3.8 Modify the code that you wrote in exercise 3.7 to calculate the properties of a circle to

calculate and use a more accurate approximation to π. Use the fact that

$$\sin\frac{\pi}{2} = 1$$

and work out the expression (using the appropriate intrinsic function) to calculate π. Store this in a new variable (perhaps called `pi`) and use this variable in your calculations of the circle. Don't forget to output the value of `pi` as a check.

Ex 3.9 Write a code that reads in the position of a point in Cartesian coordinates and converts the data to polar coordinates. Output the result in both degrees and radians (you will need to calculate π again in order to do this).

3.7 Using `CHARACTER` Data

We will not cover any more than the most basic `CHARACTER` handling as it will rarely crop up in the kind of computational modelling that is performed in physics.

We have seen some `CHARACTER` data in our `PRINT` statements (more specifically, we have seen literal constants of character type). Now we will look at declaring `CHARACTER` variables and manipulating `CHARACTER` data in our programs.

Previously, we looked at how `REAL` and `INTEGER` variable were stored in what we will call *numeric storage units*. Characters on the other hand are stored in *character storage units*. A single character storage unit will hold exactly one character. A `CHARACTER` variable consists of one or more character storage units.

Characters are taken from the FORTRAN character set which includes all the letters (in both upper and lower case), the numbers 0-9 and some additional special characters. These are shown in table 2. This gives a total of 83 characters. Other characters will almost certainly be available in any particular implementation on Fortran, and these may be used in `CHARACTER` variables and comments. However, such a program may not work on a different computer under a different implementation of Fortran.

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9			
=	+	-	*	/	(	)	,	.	’	:	!	“
%	&	;	¡	¿	?	$	space					

Table 2: The 83 characters that appear in the Fortran character set

3.7.1 Declaring `CHARACTER` Variables

Declaring `CHARACTER` variables is similar to declaring `INTEGER` and `REAL` variables with one important difference. It is necessary to declare how many characters the variable is storing (or more precisely, how many character storage units are required). This is done as follows

```
CHARACTER(LEN=length) :: var1, var2, ...
```

This declares the variables `var1, var2, ...` to have `CHARACTER` type and will have a length of *`length`* characters.

There are a couple of variations or short cuts for this, these are

```
CHARACTER(length) :: var1, var2, ...
CHARACTER*length :: var1, var2, ...
```

While these are slightly shorter, it is recommended for clarity that the full form of the statement is used.

Not all CHARACTER variable in your programs are required to have the same length. You can define CHARACTER variables with different length using multiple CHARACTER statements. E.g.,

```
CHARACTER(LEN=len1) :: var1, var2
CHARACTER(LEN=len2) :: var3, var4
```

It is possible to define variables of different length on a single line, e.g.,

```
CHARACTER(LEN=len1) :: var1, var2, var3*len2
```

Where the first two variables (var1 and var2) have length *len1* and the third (var3) has length *len2*. Again, it is recommended that, for clarity, the longer form is used.

3.7.2 CHARACTER Assignment

CHARACTER variables are assigned in a similar way to INTEGER and REAL variables, except that the string of characters to be assigned is surrounded by a pair of single quote marks ('...') or a pair of double quote marks ("..."). For example

```
str1 = 'Hello world!'
str2 = "Hello world!"
```

Ex 3.10 Consider the following program.

```
PROGRAM character_example

  IMPLICIT NONE
  CHARACTER(LEN=4) :: string1, string2
  CHARACTER(LEN=5) :: string3, string4

  string1='Stop'
  string2='Start'
  string3='Stop'
  string4='Start'

  PRINT *, '--', string1, '--'
  PRINT *, '--', string2, '--'
  PRINT *, '--', string3, '--'
  PRINT *, '--', string4, '--'

END PROGRAM character_example
```

What do you think the output will be from this program? Type the program in and run it, does it produce what you thought it would?

This brings up the issue of what happens when the string assigned to the variable is of a different length to the variable that it is being assigned to. You will have seen in exercise 3.10 that this is fairly straightforward. If the variable is longer than the string being assigned, then the rest of the variable is filled with blanks (or spaces). If the variable is shorter than the string, then the string is truncated from the right to the correct length.

3.7.3 Reading in Character Data

Character data can also be assigned by using the READ statement. This is done in the same way that integer and real data is read in, except that the string to be read must be surrounded by single or double quotation marks. There are some common situations when this rule would be annoying, so there are some exceptions to this. The quoting of character data being input into a READ statement is not required if all of the following are true:

1. the character data does not contain any value separators (e.g., blanks, commas or slashes);
2. the character data is contained within a single record or line;
3. the first non-blank character is not a single or double quotation mark
4. the leading characters are not a number followed by an asterisk (for reasons that will not be explored in this course).

If the inputted string is too long or short for the variable, it will be truncated or padded with spaces as it would for assignment.

Ex 3.11 Consider the following program.

```
PROGRAM character_example2

  IMPLICIT NONE
  CHARACTER(LEN=6) :: string1, string2, string3

! set some initial values
  string1 = ''
  string2 = ''
  string3 = ''

! prompt and read in values for the three strings
  PRINT *, 'Please enter three strings'
  READ *, string1, string2, string3

! output the three strings
  PRINT *, '--', string1, '--'
  PRINT *, '--', string2, '--'
  PRINT *, '--', string3, '--'

END PROGRAM character_example2
```

Compile and run the program and enter the following values;

(a) `'One', 'Two', 'Three'`

(b) `'One' 'Two' 'Three'`

(c) `One Two Three`

(d)
```
One Two
Three
```

(e) `'One Two', 'Three', 'Four'`

(f) `1 2 3`

(g) `'One', , three`

(h) `One /`

Try more test values until you are happy that you understand what is going on.

3.7.4 `PRINT`ing `CHARACTER` Data

We have seen in the last couple of examples that the `PRINT` statement works exactly as you might expect. The value of `CHARACTER` variables is printed without the quotation marks, but with any trailing blanks that may have been added.

3.7.5 `CHARACTER` Expressions

We have seen that with numerical variables, we can construct numerical expressions in order to perform calculations. In order to do anything useful with character variables, we would like to be able to construct character expressions.

One of the first things we can do is to concatenate two strings. This is done with the concatenation operator, `//`. So for example,

```
CHARACTER(LEN=6) :: str1, str2
CHARACTER(LEN=12) :: str3

str1 = 'Hello '
str2 = 'World!'
str3 = str1//str2
```

In this example, the values of `str1` and `str2` are concatenated to form the new string `'Hello World'` which is stored in `str3`. The usual rules for truncating/padding if the length of `str3` is different apply again.

This is the only operator provided by Fortran for character data. However, another feature is the identification of substrings. A substring is a portion of another larger string and is identified in Fortran by a pair of integers separated by a colon, contained in parentheses following a character variable, so

```
PRINT *, str3(2:5)
```

would print `'ello'` to the screen (i.e., the substring is from the second position to the fifth position of the main string). If the first integer is missing, then the substring is taken from the first character position. If the second integer is missing then the substring extends to the final character position.

Ex 3.12 Consider the following program

```
PROGRAM substring_example

  IMPLICIT NONE
  CHARACTER(LEN=16) :: str1
  CHARACTER(LEN=8) :: substr1, substr2, substr3

  str1 = 'Physics is fun!!'

  PRINT *, str1

  substr1 = str1(3:10)
```

```
    PRINT *, substr1

    substr2 = str1(:8)
    substr3 = str1(9:)
    PRINT *, substr2
    PRINT *, substr3
    PRINT *, substr2//substr3

    substr2 = 'Spanish'
    PRINT *, substr2//substr3

    str1(:7) = 'Archery'
    PRINT *, str1

    substr3(4:7) = 'hard'
    PRINT *, substr2//substr3

END PROGRAM substring_example
```

What do you think will be outputted to the screen? Type in and run the program, does it agree with what you thought? Try creating and concatenating your own strings and substrings until you understand what is going on.

In addition to the concatenation operator and substrings, there are a range of intrinsic functions for character data. A selection of these are listed in table 3.

Function	**Input**	**Output**	**Purpose**
`ACHAR(i)`	integer	character*1	Returns the character in the `i`th position of the ASCII set
`ADJUSTL(str)`	character*n	character*n	Removes all leading blanks and adds them to the end of the string
`ADJUSTR(str)`	character*n	character*n	Removes all trailing blanks and adds them to the beginning of the string
`IACHAR(c)`	character*1	integer	Returns the position in the ASCII set of the character `c`
`INDEX(str, substr)`	character*n, character*m	integer	Returns the starting position of the string `substr` within the string `str`
`LEN(str)`	character*n	integer	Returns the length of the string `str`
`LEN_TRIM(str)`	character*n	integer	Returns the length of the string `str` with all trailing blanks removed
`REPEAT(str, i)`	character*n, integer	character*(n*i)	Returns a string which is made up of the string `str` concatenated `i` times
`SCAN(str, set)`	character*n, character*m	integer	Scans the string `str` for one of the characters in `set` and returns the position of the first instance
`TRIM(str)`	character*n	character*(n-i)	Returns a string with all trailing blanks removed

Table 3: Table of selected character intrinsic functions.

Ex 3.13 Consider the following program.

```
PROGRAM character_example3

  IMPLICIT NONE
  CHARACTER(LEN=12) :: str
  INTEGER :: a_upper, a_lower, excl, spc
  INTEGER :: p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12

  a_upper = IACHAR('A') - 1
  a_lower = IACHAR('a') - 1
  excl = IACHAR('!')
  spc = IACHAR(' ')

  p1  = a_upper + 8
  p2  = a_lower + 5
  p3  = a_lower + 12
  p4  = a_lower + 12
  p5  = a_lower + 15
  p6  = spc
  p7  = a_upper + 23
  p8  = a_lower + 15
  p9  = a_lower + 18
  p10 = a_lower + 12
  p11 = a_lower + 4
  p12 = excl

  str = ACHAR(p1) // ACHAR(p2) // ACHAR(p3) // ACHAR(p4) // &
      & ACHAR(p5) // ACHAR(p6) // ACHAR(p7) // ACHAR(p8) // &
      & ACHAR(p9) // ACHAR(p10) // ACHAR(p11) // ACHAR(p12)

  PRINT *, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12
  PRINT *, str

END PROGRAM character_example3
```

Can you work out what the outputted message stored in `str` is? Type in and run this program. Did the message agree with what you thought? Modify the program so that it outputs a message of your choosing (altering the length of `str` if necessary).

Ex 3.14 Write a program that will read in a single word of arbitrary length, remove any leading and trailing blanks and return the number of letters in the word. Test your program with a selection of different words with different numbers of leading and trailing blanks.

Ex 3.15 Write a program that reads in the users first, middle and last names into three different variables, removes any unnecessary leading or trailing blanks and outputs the welcome `'Hello` *`first middle last`*`.'` (include the full stop at the end). Does your program still get the spacing correct if you omit the middle (or any of the other) names?

3.8 Initial Values and Constants

There is one additional method of assigning a value to a variable and that is to provide an initial value as part of the declaration. To do this, append an equals sign and a value on the declaration line after a variable. For example

```
REAL :: a=0.0, b=1.0, c, d, e=1E-6
INTEGER :: nmin=10, nmax=100
CHARACTER(LEN=10) :: name='undefined'
```

The initial values supplied must be either a literal constant, or a *constant expression* (i.e., an expression where all the component parts are constant).

A related issue is that of *named constants*. You may want to use some form of physical constant which will not change during the program (such as the value of π in exercise 3.8). Alternatively, you may want to define some constant for the program such as a variable called `max_cases` which wont change during a single run of your code, but you may want to change as the your problem changes (it is easier to change a single value at the top of your program rather than every instance within your code).

To define a named constant, you include the tag `PARAMETER` after the variable type. For example

```
REAL, PARAMETER :: pi=3.1415, pi_by_2=pi/2.0
INTEGER, PARAMETER :: max_cases=100
CHARACTER, PARAMETER :: filename='output.dat'
```

Notice that we use the value of `pi` to define `pi_by_2`, this is okay as we have already defined `pi` to be a constant.

4 Decision Making in Fortran

4.1 Block IF Constructs

So far all of our programs have been linear. That is, we've started at the beginning of the program and executed every statement until we've reached the end. However, in practise we may want to do something only under certain conditions, or may not do something under other conditions (for example, you may not want to take the square root of a negative number). Perhaps there is an either or situation (such as a step function ($H(x)$ which is 0 when $x < 0$ and 1 otherwise). You may even have problems when you must perform one of many actions depending on some criteria.

This is done using what is known as a `block IF construct`, an example of this is

```
IF (x>0) THEN
  rootx = SQRT(x)
END IF
```

This is the most minimal block `IF`. In the parentheses after the `IF` statement but before the `THEN` statement, we have a condition that must be satisfied (in this case, x must be greater than 0), if this is true then the action within the block `IF` will be taken (the square root will be calculated) and then the block ends with an `END IF` statement.

If you have an either or situation, you may want to include an `ELSE` statement, an example of this would be

```
IF (x>0) THEN
  stepx = 1.0
ELSE
  stepx = 0.0
END IF
```

If the condition after the `IF` statement is true, then the commands after the `THEN` statement are followed, if the condition is false, then the commands after the `ELSE` statement are followed.

If you have multiple conditions and actions, then this can be programmed as follows

```
IF (i == 1) THEN
  PRINT *, 'First case'
ELSE IF (i == ncases-1) THEN
  PRINT *, 'Penultimate case'
ELSE IF (i == ncases) THEN
  PRINT *, 'Last case'
ELSE
  PRINT *, 'Case number ',i
END IF
```

If the first condition is true (note `==` means *is equal to*), then Fortran will perform the associated action, if it is false then Fortran will examine the next condition until it finds one that is true or encounters the general `ELSE` statement. If Fortran finds a condition that is true and performs the associated actions, it will then jump to the end of the block `IF` without evaluating any of the other conditions (even if some of them are also true). The general `ELSE` statement is optional, if it is not present and none of the above conditions are true, then the block `IF` will perform no action.

4.1.1 Indentation

You may have noticed that the action statements have been indented from the `IF` and `ELSE` statements. This is not necessary for Fortran itself, however it is a very popular programming convention that makes the code easier to read. There are a variety of different programming blocks (like the `IF` block and the `PROGRAM` block) that may be nested within one another. Using sensible indentation allows the programmer (and other programmers) to easily see which `IF`s, `ELSE`s and `END`s are associated with one another.

Consider the following bit of code, first without indentation

```
IF (i == 1) THEN
IF (j == 1) THEN
PRINT *, 'This is the top-left corner'
ELSE IF (j == jmax) THEN
PRINT *, 'This is the bottom-left corner'
ELSE
PRINT *, 'This is the left edge'
END IF
ELSE IF (i == imax) THEN
IF (j == 1) THEN
PRINT *, 'This is the top-right corner'
ELSE IF (j == jmax) THEN
PRINT *, 'This is the bottom-right corner'
ELSE
PRINT *, 'This is the right edge'
END IF
ELSE
IF (j == 1) THEN
PRINT *, 'This is the top edge'
ELSE IF (j == jmax) THEN
PRINT *, 'This is the bottom edge'
ELSE
PRINT *, 'This is in the middle'
END IF
END IF
```

And now with the indentation

```
IF (i == 1) THEN
  IF (j == 1) THEN
    PRINT *, 'This is the top-left corner'
  ELSE IF (j == jmax) THEN
    PRINT *, 'This is the bottom-left corner'
  ELSE
    PRINT *, 'This is the left edge'
  END IF
ELSE IF (i == imax) THEN
  IF (j == 1) THEN
    PRINT *, 'This is the top-right corner'
  ELSE IF (j == jmax) THEN
    PRINT *, 'This is the bottom-right corner'
  ELSE
```

```
      PRINT *, 'This is the right edge'
    END IF
  ELSE
    IF (j == 1) THEN
      PRINT *, 'This is the top edge'
    ELSE IF (j == jmax) THEN
      PRINT *, 'This is the bottom edge'
    ELSE
      PRINT *, 'This is in the middle'
    END IF
  END IF
```

In the first code, it is not immediately clear which `IF`s, `ELSE`s, `END`s and actions belong with each block and what the flow of choices and actions are. In the second code, the flow of the code is much easier to see.

It is up to you to use as much or little visual formatting as you wish, but indentations of 2-3 spaces (but be consistent) are strongly encouraged.

Ex 4.1 Write a program using the block `IF` construct to read in an integer, and output whether the number is positive or negative. Compile and run for several test values. What happens if you type in `0`? If necessary, modify you block `IF` to handle this case as well.

Ex 4.2 Write a program using nested block `IF` constructs to read in two real numbers and evaluate the following function.

$$f(x,y) = \begin{cases} x+y & \text{if } x \geq 0, y \geq 0 \\ x-y & \text{if } x \geq 0, y < 0 \\ -x+y & \text{if } x < 0, y \geq 0 \\ -x-y & \text{if } x < 0, y < 0 \end{cases}$$

Don't forget to output your result. Can you also write this as a Fortran expression without using a block `IF` construct?

4.2 Logical Variables

Clearly, an important part of the block `IF` is the condition. So far we have seen tests for *equal to* (`==`), *greater than* (`>`) and *less than* (`<`). To fully understand these conditions, we must first look at `LOGICAL` type variables.

In Fortran, `LOGICAL` variables are (unsurprisingly) declared by the statement

```
LOGICAL :: var1, var2, ...
```

They are basically yes-no, or true-false variables and can take one of two values which are

```
var1 = .TRUE.
var1 = .FALSE.
```

Note, the full stops are part of the expression.

The real power of `LOGICAL` variables is Fortran's ability to perform `LOGICAL` expressions. We have already seen three `LOGICAL` expressions (`==`, `<` and `>`), these are known as *relational operators* and there are six of these in total, although each can be written in one of two ways. The complete list is:

Operator	Alternate Form	Result
`a < b`	`a .LT. b`	`.TRUE.` if `a` is less than `b`
`a <= b`	`a .LE. b`	`.TRUE.` if `a` is less than or equal to `b`
`a > b`	`a .GT. b`	`.TRUE.` if `a` is greater than `b`
`a >= b`	`a .GE. b`	`.TRUE.` if `a` is greater than or equal to `b`
`a == b`	`a .EQ. b`	`.TRUE.` if `a` is equal to `b`
`a /= b`	`a .NE. b`	`.TRUE.` if `a` is not equal to `b`

Either form of the operator is fine to use in logical expressions, although it is good practise to be consistent with whichever form you choose.

These operators can be used with literal constants, variables and other expressions, for example

```
var1 = 2.0 >= 1.0    ! var1 is .TRUE.
var2 = 3 == 4        ! var2 is .FALSE.
var3 = x .LT. 0      ! .TRUE. if x is less than 0
var4 = i /= j
var5 = x*y .LT. 0
var6 = b**2.0 .GE. 4.0*a*c
```

You may notice that each of these are effectively one expression, and the order in which the operators are carried out may be important. This rule is simple, all arithmetic operators have a higher priority than the relational operators, and so are performed first. If in doubt then add some brackets, this may also improve the legibility of your code.

In addition to the relational operators, there are also five *logical operators*. Where the relational operators acted on real or integer (or even character) variables, the logical operators act on logical variables. The operators are

Operator	Priority
.NOT.	highest
.AND.	
.OR.	
.EQV., .NEQV.	lowest

The order of priority is shown, and they all have lower priority than the relational operators. The results given by these operators are

L1	L2	L1.AND.L2	L1.OR.L2	L1.EQV.L2	L1.NEQV.L2
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

L1	.NOT.L1
.TRUE.	.FALSE.
.FALSE.	.TRUE.

These operators may be used in a similar way to operators we've already covered. For example

```
var1 = L1.AND.L2 .OR. L3.AND.L4
var2 = x<0 .OR. x>1
var3 = .NOT. (i==1 .OR. i==4)
var4 = b**2.0 .GE. 4.0*a*c .AND a*c .GT 0.0
```

Again, if you're not sure about the priority, or if you wish to improve the clarity of the code, feel free to add brackets. So `var4` might be better written

```
var4 = (b**2.0 .GE. 4.0*a*c) .AND (a*c .GT 0.0)
```

Ex 4.3 For each of the cases below, work out (by hand) what the eventual output will be.

(a)
```
x = 1.0
i = 4
var1 = x>0.0 .AND. i<5
PRINT *, var1
```

(b)
```
x = -2.5
y = 3.1
i = 2
j = 5
var1 = x*y > 0 .NEQV. i/j /= 0
PRINT *, var1
```

(c)
```
x = 1.5
y = 1.6
i = 2
j = -2
var1 = .NOT. x+y > 3.0 .OR. i-j > 3
var2 = .NOT. (x+y > 3.0 .OR. i-j > 3)
PRINT *, var1, var2
```

(d)
```
x = 3.2
y = -4.7
i = 1
j = 5
var1 = x>0 .OR. y>0 .AND. i==1 .OR. j==1
var2 = (x>0 .OR. y>0) .AND. (i==1 .OR. j==1)
PRINT *, var1, var2
```

Type in and run each case (all in a single program if you prefer) to check your answers.

4.3 Return to the Block `IF` Construct

In section 4.1 we took an informal look at the block `IF` construct. Now we know about `LOGICAL` expressions, we will look at it a bit more formally. The first line of the block `IF` construct is

```
IF (condition) THEN
```

where *`condition`* is a `LOGICAL` entity. This could be a `LOGICAL` literal constant (though this usually a bit pointless), a `LOGICAL` variable or (more usually) a `LOGICAL` expression. If the `LOGICAL` entity evaluates to `.TRUE.` then the statements following the `THEN` statement will be performed. These statement (there may be more than one line) are terminated by an `ELSE IF`, `ELSE` or `END IF` statement.

If the `LOGICAL` entity is `.FALSE.` then Fortran will proceed sequentially through any `ELSE IF` statements until a `.TRUE.` result is found or an `ELSE` statement is reached (Fortran will perform all statements under the `ELSE` statement) or an `END IF` statement is found. The *`condition`* under any `ELSE IF` statements follow the same rules as for a standard `IF` statement.

The `END IF` statement must always be present somewhere after an `IF` statement. There may be any number of `ELSE IF` statements and a maximum of one `ELSE` statement. An `ELSE IF` statement may not follow an `ELSE` within a single level of a block `IF` construct.

Ex 4.4 Write a code to evaluate the function in exercise 4.2 using a single block `IF` construct (rather than nested ones).

Ex 4.5 Consider the code in section 4.1.1. Modify this code so that it uses a single level block `IF` construct rather than nested ones. Put this in a program that sets `imax` and `jmax` as parameters and asks for input of `i` and `j`. Add options to the if statement to check that `i` and `j` are in the specified range (i.e., `1<i<imax` and `1<j<jmax`).

4.4 The CASE Construct

Fortran 90 offers another decision making construct called the *CASE construct*. This is used when there is only one basic test, but there may be many different options. An example of this might be offering the user a menu with several different options and asking the user to select the one they want, the program will then do different things depending on the option selected.

The basic syntax is

```
SELECT CASE (case_expression)
CASE (case_selector1)
  ...
CASE (case_selector2)
  ...
CASE (case_selector3)
  ...
.
.
.
CASE DEFAULT
  ...
END SELECT
```

The *case_expression* is a integer, character or logical variable or expression (real expressions are prohibited for this construct) which is evaluated and the result used for comparison with the different *case_selector*s.

The *case_selector*s determine which statements are executed depending on how they evaluate with the *case_expression*. The *case_selector* may take one of four forms

```
case_value
low_value:
:high_value
low_value:high_value
```

or a comma-separated list of any combination of these. The meaning of the four alternatives is as follows:

- If the *case_selector* takes the form of *case_value* then the associated block of code is executed if *case_expression* `==` *case_value* or *case_expression* `.EQV.` *case_value* for logical expressions;
- If the *case_selector* takes the form of *low_value*`:` then the associated block of code is executed if *low_value* `<=` *case_expression*;

- If the *case_selector* takes the form of :*high_value* then the associated block of code is executed if *case_expression* <= *high_value*;
- If the *case_selector* takes the form of :*high_value* then the associated block of code is executed if *low_value* <= *case_expression* .AND. *case_expression* <= *high_value*.

Note that only the *case_value* makes sense for logical expressions.

The CASE DEFAULT statement is an optional catch all that can contain a block of code to be run if no other statement is matched. This is similar to the ELSE statement in the block IF construct.

As an example, consider the following code.

```
! read in the date
  PRINT *, 'Enter the month as a number'
  READ *, month

! print season
  SELECT CASE (month)
  CASE (11, 12, 1:3)
    PRINT *, 'It is the winter'
  CASE (4:5)
    PRINT *, 'It is the spring'
  CASE (6:8)
    PRINT *, 'It is the summer'
  CASE (9, 10)
    PRINT *, 'It is the autumn'
  CASE DEFAULT
    PRINT *, 'You have not entered a valid month'
  END SELECT
```

This shows how the CASE construct might work in practise.

Ex 4.6 Write a program to read in the month as a number and use a CASE construct to print out the name of the month.

Ex 4.7 Write a program to read in two real numbers, print out a list of options of what to do with the two numbers (add them together, subtract the second from the first, multiply them, divide the first by the second or raise the first to the power of the second) and read in the answer, then use a CASE construct to perform (and output) the selected option.

5 Repeating Parts of your Program

Often it is necessary to repeat steps of your program. In fact, many techniques in computational mathematics/physics rely on being able to iterate through the same set of instructions. For example, suppose you wanted to output a times table, you could probably write out the first dozen or so lines of the table, but what if you wanted to output 100 lines or 1000. Typing each line out individually would be very time consuming and tedious. Much better to write the calculation in a generic way and iterate through as many times as required.

5.1 Block DO Constructs

Fortran 90 has a very powerful, yet simple facility for looping through parts of your code, this is called the *block DO construct*. The syntax for this block is

```
DO count = initial, final, incr
  block of statements
  .
  .
  .
END DO
```

where *count* is an integer variable, *initial* is an integer literal constant, variable or expression that indicates the first value that *count* takes, *final* is an integer literal constant, variable or expression that indicates the maximum (or minimum) value that *count* takes, and *incr* which is the amount that *count* increases (or decreases) by in each step.

The DO statement may also take the following forms

```
DO count = initial, final
  block of statements
  .
  .
  .
END DO
```

where *count* is increased by 1 each time, and

```
DO
  block of statements
  .
  .
  .
END DO
```

where the loop will continue indefinitely. This last case may initially seem like a strange thing to want to do, after all, we want our program to finish at some point. However, as we shall see later there are alternative ways to break out of a block DO construct.

Ex 5.1 Consider the following program.

```
PROGRAM timestable

  IMPLICIT NONE
  INTEGER :: tableno, i
```

```
! choose to do seven times table
  tableno=7

! output times table
  DO i=1,12
    PRINT *, i, ' times ', tableno, ' is ', i*tableno
  END DO

END PROGRAM timestable
```

Type in and run this program. Modify the program so that it outputs more or less than 12 lines and does different table numbers. Can you modify the program so that the user can input the table number and number of lines that it outputs?

Note that in the program in exercise 5.1 we indented the block of statements within the block `DO` in the same way that we did for blocks of code in the block `IF` construct. Again, this is good practise and makes the code easier to read and it is suggested that you follow the same convention within your own codes.

Ex 5.2 The Fibonacci sequence is calculated as follows.

1. The first and second Fibonacci numbers are both equal to 1.
2. The *i*th Fibonacci number is calculated by adding the previous two Fibonacci numbers, i.e.,

$$F_i = F_{i-2} + F_{i-1}$$

So the first eight numbers in the Fibonacci sequence are 1, 1, 2, 3, 5, 8, 13, 21.

Write a program that calculates the first 20 numbers in the Fibonacci sequence using a block `DO` construct.

Both of these exercises have used the most simple count-controlled loop, where the *`count`* is incremented by 1 each time. What happens when we use the more advanced count-controlled loop which includes the *`incr`* increment?

Consider the following code

```
DO i=1,9,2
  PRINT *, i
END DO
```

When this is run, it simple outputs the values that `i` takes in each iteration of the loop. In this case it outputs the integers `1`, `3`, `5`, `7`, `9`. So what happens if we modify the code as follows

```
DO i=1,10,2
  PRINT *, i
END DO
```

In this case, the maximum is 10, but `i` can not take that value with increments of 2. In this case, Fortran 90 stops iterating at the highest value it can without exceeding the maximum. So the output would be the integers `1`, `3`, `5`, `7`, `9` again.

Ex 5.3 Write a program that performs the following loops and prints out the values that `i` may take in each iteration.

(a) `DO i=1,10`

(b) `DO i=1,10,1`

(c) `DO i=25,60,5`

(d) `DO i=8,24,3`

(e) `DO i=1,2,3`

(f) `DO i=3,2,1`

(g) `DO i=-21,21,7`

(h) `DO i=10,1,-1`

(i) `DO i=21,-21,-7`

Are they what you expected? What is the value of `i` immediately after the block `DO` construct has ended? Try some different loops of your own devising until you understand what it going on.

5.2 More Advanced Loops

5.2.1 The `EXIT` Statement

The count-controlled loops that we have seen so far will iterate a fixed number of times depending on the count range in the `DO` statement. Sometimes the number of times that we want to iterate a method is undetermined at the start of a loop. An example of this may be calculating the position of a projectile, we want to calculate the position until the projectile hits the ground (i.e., $y = 0$).

To do this, we use the `EXIT` statement. When used within a block `DO` construct, this will immediately transfer control of the program to the first statement after the `END DO` statement. The `EXIT` statement is almost always used with an `IF` or `CASE` statement.

Consider the following example, we want to calculate Fibonacci numbers again, but this time instead of stopping at the 20th Fibonacci number, we want to display all the Fibonacci numbers under 1000 (so we want to stop before the first Fibonacci number greater than 1000 is outputted).

We can do this with the following program

```
PROGRAM fibonacci2

  IMPLICIT NONE
  INTEGER :: f_old1=0, f_old2=0, f_new, i

! method to calculate the Fibonacci numbers under 1000
! do the initial stuff first
  f_old1 = 1
  f_old2 = 1
  PRINT *, 'The ', 1, 'st Fibonacci number is ', f_old1
  PRINT *, 'The ', 2, 'nd Fibonacci number is ', f_old2
  i = 3

! now perform the DO loop until f_new>1000
  DO
    f_new = f_old1 + f_old2
    IF (f_new > 1000) EXIT
    PRINT *, 'The ', i, 'th Fibonacci number is ', f_new
```

```
    f_old1 = f_old2
    f_old2 = f_new
    i = i + 1
  END DO

END PROGRAM fibonacci2
```

In this case we have an indefinite `DO` loop which contains an `IF` statement to check our criteria. If the criteria is met, then the `EXIT` statement immediately causes Fortran to exit the `DO` loop.

Note, in this case we have used a simple form of the `IF` statement. In cases where we just want to execute a single command if some condition is true, and we do not want any `ELSE IF` or `ELSE` statements, then we can drop the `THEN` and replace it with the statement we wish to execute.

5.2.2 When Indefinite Loops Go Bad

There is one thing to be cautious of here, when we perform the `IF` statement, we want to be sure that the condition will be met at some point. For example, if we had written the statement

```
IF (f_new == 1000) EXIT
```

we would have been in trouble. 1000 itself is not a Fibonacci number, so the condition would never be met and the loop would go on infinitely (or at least until we killed the program externally). So if possible, we always want a condition that we know will be fulfilled at some point.

If we don't know that our condition will be fulfilled, then it is a good idea to program ourselves some kind of get out clause. One way to do this is to decide that there will be a maximum number of iterations that we will try before we give up. For example

```
DO count=1, max_iterations
  .
  .
  .
  IF (condition) EXIT
  .
  .
  .
END DO
```

In this case we set the variable `max_iterations` to be much higher than we would expect to be needed, if *`condition`* is met then Fortran exits from the loop, if it is never met, then Fortran will eventually exit from the loop anyway (though not necessarily with the desired result).

Ex 5.4 The Secant algorithm is a method of finding the roots of an equation (i.e., where $f(x) = 0$). The theory behind the algorithm is beyond the scope of this course, but the basic algorithm is as follows.

1. Set x_1 and x_2 equal to two different (but close) values, preferably near the root you want to find.
2. Calculate further values of x using the following formula

$$x_n = x_{n-1} - f(x_{n-1}) \left[\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \right]$$

3. Iterate until one of the following conditions is true

$$\begin{aligned} |f(x_n)| &< tol_1 \\ |x_n - x_{n-1}| &< tol_2 \end{aligned}$$

Consider the following equation

$$x^3 - 5.12x^2 - 74.0763x + 106.87383$$

which has three real roots. Write a program to perform the Secant method on this equation. Use the following tolerances, $tol_1 = 0.0001$, $tol_2 = 0.0001$ and for the initial guesses try $x_1 = 0.0$ and $x_2 = 0.5$. Output the iterate number, the value of x and the value of $f(x)$ for each iteration.

Do you know that the exit condition for the loop iterations will eventually be met? If not program in a get out clause.

Try different initial guesses to find all three roots of the equation (hint, all three lie between -20 and 20).

5.2.3 The `CYCLE` Statement

The `CYCLE` statement is similar in usage to the `EXIT` command, but instead of exiting from the block `DO` construct, it halts execution of the block of code and returns control to the top of the block `DO` loop to begin the next iteration. This causes any counter attached to the `DO` statement to be increased accordingly.

For example, consider the following bit of code.

```
DO i=-5,5,2
  IF (i<0) CYCLE
  PRINT *, 'The square root of ', i, ' is ', SQRT(REAL(i))
END DO
```

In this case we want to take square roots of `i`, but not if `i` is negative, so for those instances we use the `CYCLE` statement to go to the next iteration.

This is a trivial example and we could just as easily have put the `PRINT` statement in a block `IF` construct for when $i \geq 0$. However, in more complex programs, there may be a lot more statements in the block `DO` construct and using the `CYCLE` statement may be more appropriate.

Ex 5.5 Modify the program that you wrote in exercise 3.8 where you calculated different properties of a circle. Enclose the code where you read in a radius and output the different properties in a block `DO` loop so that you are prompted to do this five times. Put in a check after reading in the the radius to see if it is negative. If it is output an error message and `CYCLE` back to the top of the block `DO`.

5.2.4 The `STOP` Statement

Strictly speaking, the `STOP` is not specific to block `DO` constructs and may appear anywhere in your program. This statement has the effect that it stops execution of your program immediately. This has the same effect as transferring control of your program directly to the `END PROGRAM` statement.

There are a variety of cases where you may want to use the `STOP` statement. If your program was going badly wrong, you may have a check in there to stop execution of your program.

For example, in the Secant method, you monitored the value of $f(x)$ to see if it was close to 0 (i.e., was $|f(x)| < tol_1$), you could also monitor it to see if it becomes too big (i.e., $|f(x)| > tol_3$ where $tol_3 = 10\ 000$ say) and stop execution if it does (this might indicate that the method has been programmed incorrectly). You may want to keep an eye on x, if it goes out of the range $[-1000, 1000]$ you may decide that it is going out of range and stop execution (this might indicate a bad choice of starting points, or the function has no roots).

If you are trying to debug code, you may want to scrutinise a particular part of the code and stop execution directly afterwards, rather than finish running a code that does not work (and may take some time to finish).

For example, consider the timestable program again. We can put in some checks that the user types in sensible information and exit with an error message if they do not.

```
PROGRAM timestable

  IMPLICIT NONE
  INTEGER :: tableno, nlines, i

! inputs the table no and number of lines to display
  PRINT *, 'Enter which number timestable you would like'
  READ *, tableno

  PRINT *, 'Enter the number of lines you would like to display'
  READ *, nlines

! check that the number of lines is positive
  IF (nlines <= 0) THEN
    PRINT *, 'The number of lines must be positive'
    STOP
  ENDIF

! output times table
  DO i=1,nlines
    PRINT *, i, ' times ', tableno, ' is ', i*tableno
  END DO

END PROGRAM timestable
```

5.3 Nested Block DO Constructs

5.4 Using Nested Block DO Constructs

It is perfectly acceptable to nest one block `DO` within another, in fact you could have multiple nestings of block `DO`s. For example

```
PROGRAM nested_do_example

  IMPLICIT NONE
  INTEGER, PARAMETER :: imax=5, jmax=5
  INTEGER :: i, j
```

```
  DO j=1, jmax
     DO i=1,imax

        IF (i == 1 .AND. j == 1) THEN
           PRINT *, 'The top-left corner is at     (', i, j, ')'
        ELSE IF (i == 1 .AND. j == jmax) THEN
           PRINT *, 'The bottom-left corner is at  (', i, j, ')'
        ELSE IF (i == imax .AND. j == 1) THEN
           PRINT *, 'The top-right corner is at    (', i, j, ')'
        ELSE IF (i == imax .AND. j == jmax) THEN
           PRINT *, 'The bottom-right corner is at (', i, j, ')'
        END IF

     END DO
  END DO

END PROGRAM nested_do_example
```

One thing to keep an eye on with nested block `DO` loops is the number of times your program is doing a block of statements. If you have a single loop, then you are executing the nested block of statements `imax` times. In the case above with a nested loop, then the number of times the nested block of statements is executed is `imax`$\times$`jmax`. Similarly, if there were two nested loops the number of times the nested block of statements is executed is `imax`$\times$`jmax`$\times$`kmax`.

The problem with this is the length of time it takes to perform the whole block `DO` construct, this is equal to the time it takes to do the block of statements once is multiplied by the number of times the block is executed. So if you have a block of statements that takes 0.5 seconds to perform and `imax`$=$`jmax`$=$`kmax`$= 100$. For a single loop the block `DO` would be completed in about `imax`$\times 0.5 =$ 50 seconds. For a nested loop it takes about `imax`$\times$`jmax`$\times 0.5 = 5\ 000$ seconds or 1.4 hours. For two nested loop it takes about `imax`$\times$`jmax`$\times$`kmax`$\times 0.5 = 500\ 000$ seconds or 5.8 days.

Clearly, if you are using nested loops then it is important that the block of code within the nested loops is efficient.

5.4.1 `EXIT` and `CYCLE` Statements in a Nested Block `DO`

Consider the following bit of code.

```
DO j=1, jmax
  .
  .
  .
  DO i=1, imax
    .
    .
    .
    IF (condition) EXIT
    .
    .
    .
  END DO
  .
```

```
  .
  .
END DO
```

In this code, which block `DO` does the `EXIT` statement break out of? Is it the inner block `DO` (i.e., `DO i=1, imax`) or the outer block `DO` (i.e., `DO j=1, jmax`). In practise, when you're writing a code, you may want to choose which block `DO` you wish to break out of (there will be times when you only want to exit the inner block and times when you want to exit the outer block).

To solve this problem, you can give your blocks names, then when you wish to `EXIT` or `CYCLE`, you can specify by name which block you are referring to. For example,

```
outer: DO j=1, jmax
  .
  .
  .
  inner: DO i=1, imax
    .
    .
    .
    IF (condition1) EXIT inner
    IF (condition2) EXIT outer
    IF (condition3) CYCLE inner
    IF (condition4) CYCLE outer
    .
    .
    .
  END DO inner
  .
  .
  .
END DO outer
```

In this piece of code, the block `DO` constructs are labelled (`outer` and `inner` respectively), and when the `EXIT` and `CYCLE` statements are called, they are followed by the name of the block `DO` to which they refer. Note that the `END DO` statements are also followed by the block name to which they refer.

Note, in the absence of a label following the `EXIT` or `CYCLE` statement, then the statement refers to the innermost block `DO` construct which contains the statement. However, in these situations you should always label your loops, `EXIT` and `CYCLE` statements to improve the readability of your code.

Ex 5.6 In this exercise we will modify the Secant method from exercise 5.4. Instead of manually trying different initial values of x_1 and x_2, we want to loop through a progression of values from $x_1 = -20$, $x_2 = -19.5$ to $x_1 = 19.5$, $x_2 = 20$, incrementing each initial value by 0.5 each time.

This will require a nested block `DO` where the initial values are varied in the outer loop and the Secant method is performed in the inner loop.

The Secant method can iterate until either

$$\begin{aligned} |f(x_n)| &< tol_1 \\ |x_n - x_{n-1}| &< tol_2 \end{aligned}$$

when you want to `EXIT` from the Secant method and output the result. Or

$$\begin{aligned} |f(x_n)| &> tol_3 \\ |x_n - x_{n-1}| &> tol_4 \end{aligned}$$

when the method is deemed not to be converging and you want to `CYCLE` to the next set of initial values. Output a message to say that the method is diverging and it is cycling to the next value.

Use the following tolerances, $tol_1 = 0.0001$, $tol_2 = 0.0001$, $tol_3 = 1000.0$, $tol_4 = 1000.0$.

Do you find the three roots that you found in exercise 5.4?

5.4.2 Naming Block `IF` Constructs

You can also assign names to block `IF` constructs (nested or otherwise) and `CASE` statements as follows,

```
name1: IF (condition1) THEN
  .
  .
  .
END IF name1
```

and

```
name2: SELECT (case_expression)
CASE (case_range_1)
  .
  .
  .
CASE (case_range_2)
  .
  .
  .
END SELECT name2
```

The naming of block `IF` constructs has no useful purpose from a coding point of view, but it may make your code more readable. You may use this as you see fit.

6 Introduction to Arrays

6.1 The Concept of Arrays

So far in this course we have restricted ourselves to manipulating single values (e.g., `INTEGER`s and `REAL`s. In scientific computing we often want to manipulate sets of data, such as a set of readings from an experiment, vectors and matrices, or a set of values generated from a numerical code (e.g., positions of a projectile at different times in flight).

Fortran 90 has powerful processing features to manipulate such sets of data (known as arrays) and this is where the real strength of Fortran 90 as a programming language lies.

When we have used variables in the previous chapters we have used them to store a single number, but it would also be useful if we could store a set of values under a single variable name and refer to individual elements by index (in mathematics we can write a vector $\mathbf{x}$ and refer to individual elements as $x_1, x_2, ..., x_n$).

In Fortran we can do this by declaring a variable as an array of a fixed length. This reserves space in memory to store all the elements of the array as well as some space to describe the data (for example, whether it's `REAL` or `INTEGER`, what the variable name is, and so on). So going back to our odometer example, a single value might be stored as follows,

x							Real
0	6	4	7	2	2	6	3

Where an array containing five numbers might be stored like this,

x		n=5					Real
0	6	4	7	2	0	2	3
0	2	7	2	1	9	5	8
9	8	3	9	3	6	0	5
0	1	2	5	7	4	2	8
9	9	8	4	4	8	4	1

So there is one line that tells you about the data, and five odometers below to store the data. If we want to know what the value of x_3, we can look at the third odometer and see that $x_3 = 0.393605 \times 10^{-2}$.

6.2 Using Arrays

6.2.1 Array Declaration

Arrays can be declared by using the `DIMENSION` statement within a variable declaration. This statement allows you to specify the number of elements in your array, and is used as follows

```
INTEGER, DIMENSION(10) :: a, b, c
REAL, DIMENSION(15) :: x, y, z
```

The first line declares `INTEGER` arrays with 10 elements and the second line declares `REAL` arrays with 15 elements.

There are shorthands in array declaration that you will sometimes see such as,

```
INTEGER :: a(10), b(20), c(30)
REAL, DIMENSION(50) :: x, y, z(60)
```

It is recommended for clarity that you do not use these shorthands and that you declare arrays of different sizes on different lines, e.g.,

```
REAL, DIMENSION(50) :: x, y
REAL, DIMENSION(60) :: z
```

By default, the array subscripts will begin at 1 and end at the number specified in the `DIMENSION` statement (i.e., indexing will be from 1 to 50 for `x` above). Sometimes it is useful to have a non standard range of subscripts, you can do this by the declaration

```
DIMENSION(lower_bound:upper_bound)
```

the index will then start at *`lower_bound`* and end at *`upper_bound`*. For example

```
INTEGER, DIMENSION(6:25) :: a
REAL, DIMENSION(0:49) :: x
REAL, DIMENSION(-30:30) :: y
```

The first case declares an `INTEGER` array with 20 elements indexed from 6 to 25; the second case declares a `REAL` array with 50 elements indexed from 0 to 49; and the third case declares a `REAL` array with 61 elements indexed from -30 to 30.

6.2.2 Array Constants and Initial Values

Arrays may be assigned and addressed as a whole or individual elements can be addressed using array subscripts. To address individual elements of an array, you follow the variable name with the subscript in brackets, e.g., `a(i)`. So you might create and initialise an array as follows,

```
INTEGER, DIMENSION(5) :: arr

arr(1) = 1
arr(2) = 2
arr(3) = 3
arr(4) = 4
arr(5) = 5
```

Or (in this case) more conveniently with

```
INTEGER, DIMENSION(5) :: arr
INTEGER :: i

DO i=1,5
  arr(i) = i
END DO
```

For smaller arrays, it would be more convenient to have a way to initialise them in one line. This can be done using an *array constant* construct. This takes the form

```
arr = (/ value1, value2, value3, ... /)
```

For example

```
INTEGER, DIMENSION(5) :: arr
arr = (/ 1, 2, 3, 4, 5 /)
```

The values in the array are contained within the delimiters `(/` and `/)`. The array constant must have the same number of values as required to fill the variable on the left-hand side of the `=` sign. The array

constant may also be used in declaration lines,

```
INTEGER, DIMENSION(5) :: arr = (/ 1, 2, 3, 4, 5 /)
```

There is an additional shorthand that can be used in array constants for defining array constants. This is a compact form of the `DO` loop and takes the form

```
(value_expression, implied_do_control)
```

For example

```
arr = (/ (i, i=1,5) /)
```

This loops through the values `i=1` to `i=5` and creates an array of all these values. You may use different expression as the *value_expression*, such as

```
arr = (/ (0, i=1,5) /)          ! gives (/ 0, 0,  0,  0,  0 /)
arr = (/ (i*2, i=1,5) /)        ! gives (/ 2, 4,  6,  8, 10 /)
arr = (/ (i**2 + 4, i=1,5) /)   ! gives (/ 5, 8, 13, 20, 29 /)
```

The more advanced form of the *implied_do_control* that was seen in section 5.1 where there was an increment option is also allowed in this context, e.g.,

```
arr = (/ (i, i=2,10,2) /)    ! gives (/ 2, 4, 6, 8, 10 /)
arr = (/ (i, i=5,1,-1) /)    ! gives (/ 5, 4, 3, 2,  1 /)
```

The usual behaviour discussed in section 5.1 also applies here.

You may use the implied `DO` construct as only part of the array, e.g.,

```
arr = (/ -1, (0, i=2,4), 1 /)        ! gives (/ -1, 0, 0, 0, 1 /)
arr = (/ -1, (0, i=1,3), 1 /)        ! gives (/ -1, 0, 0, 0, 1 /)
arr = (/ (0, i=1,2), (i, i=1,3) /)   ! gives (/  0, 0, 1, 2, 3 /)
```

Finally, one implied `DO` construct may be contained within another to create a nested loop (or many nested loops if desired). For example,

```
arr = (/ ( (i, i=1,5), j=1,5) /)
arr = (/ ( (i+j, i=1,5), j=1,5) /)
arr = (/ ( (MIN(i,j), i=1,5), j=1,5) /)
arr = (/ ( -1, (0, i=1,3), 1, j=1,5) /)
```

Ex 6.1 Write a program that declares, initialises and outputs (you can output an entire array using `PRINT *, arr`) the following arrays:

(a) A 21 element array containing the numbers from -10 to 10.

(b) An 11 element array containing the only the even numbers from -10 to 10.

(c) A 37 element array containing an equally spaced sequence of real numbers from 0 to 2π.

(d) Two 50 element arrays, the first contains the sequence 0-9 repeated 5 times, the second has 10 elements with value 0, 10 elements with value 10, 10 elements with value 20, 10 elements with value 30 and 10 elements with value 40. Also output the sum of the two arrays (`arr1+arr2`).

Hint, you may need to use nested implied `DO`s. The sum of the two matrices should give the numbers 0-49.

6.2.3 Input and Output with Arrays

Before we start to use arrays, it would be useful to know how to input an array of data and output arrays of data. There are three possible ways of inputting or outputting data depending on whether we are interested in individual elements of an array, a range of elements from the array, or a complete array.

- Array elements (e.g., `arr(1)`) are just scalar variables (single numbers) and can be used in expressions as such.
- Part of an array may be used in input and output lists by using an implied `DO` construct in a similar way as in array initialisation.
- Whole arrays may appear in input and output lists and refer to the complete array. We saw this in exercise 6.1.

So for example,

```
PRINT *, arr(1), arr(5)
READ *, (arr(i), i=1,9,2)
READ *, arr
PRINT *, arr(1), (arr(i), i=2,8,2), arr(9)  !
```

The first case outputs the 2 values in `arr(1)` and `arr(5)`; the second case inputs 5 values and stores them in `arr(1)`, `arr(3)`, `arr(5)`, `arr(7)` and `arr(9)`; case 3 inputs enough values to fill `arr`; and case 4 outputs 6 values, `arr(1)`, `arr(2)`, `arr(4)`, `arr(6)`, `arr(8)` and `arr(9)`.

The usual rules for the `READ` statement apply, i.e., blank values leave the array element unchanged, if too many values are entered then the extra ones are ignored, and so on.

One other useful thing is that the control values for an implied `DO` statement may themselves be inputted in the same statement, e.g.,

```
READ *, nitems, (arr(i), i=1,nitems)
```

Though doing so should be done with care. In the above example we could type in a value for `nitems` that is larger than the size of the array `arr` (this will cause your program to crash). It may be better to put in a check such as

```
READ *, nitems
IF ((nitems >= 1) .AND. (nitems<=maxitems)) THEN
  READ *, (arr(i), i=1,nitems)
ELSE
  ! handle error
END IF
```

6.2.4 Arrays in Expressions

In order to make use of arrays, we want to be able to use them in arithmetic expressions. There are two ways in which we can do this. The first is to address each element of the array individually, perhaps within a block `DO` construct as follows

```
REAL, DIMENSION(20) :: a, b, c
.
.
.
```

```
DO i=1,20
  a(i) = b(i) + c(i)
END DO
```

You can also perform operations on arrays as a whole, for example

```
REAL, DIMENSION(20) :: a, b, c
.
.
.
a = b + c
```

This is allowed so long as all of the arrays are *conformable*. The rules for conformability are as follows:

- Two arrays are conformable if they have the same size and shape;
- A scalar (either variable or literal constant) is conformable with any array;
- All intrinsic operations are defined between two conformable objects.

So examples of other operations are as follows.

```
REAL, DIMENSION(20) :: a, b, c
REAL :: x

a = b - c     ! = (/  b(1)-c(1), b(2)-c(2), ... /)
a = b*c       ! = (/  b(1)*c(1), b(2)*c(2), ... /)
a = b/c       ! = (/  b(1)/c(1), b(2)/c(2), ... /)
a = 2.0*b     ! = (/  2.0*b(1), 2.0*b(2), ... /)
a = c/5.0     ! = (/  c(1)/5.0, c(2)/5.0, ... /)
a = x*b       ! = (/  x*b(1), x*b(2), ... /)
a = b**2.0    ! = (/  b(1)**2.0, b(2)**2.0, ... /)
a = c + 2.5   ! = (/  c(1)+2.5, c(2)+2.5, ... /)
a = 0.0       ! = (/  0.0, 0.0, ... /)
```

When an operation acts on two arrays, then it acts on each in an element by element basis (as if in a block `DO`). When an operation acts on an array and a scalar, then the scalar is treated as an array of same size with every element taking the value of the scalar.

Note that this gives us an easy way of initialising arrays where we want all elements to take the same value, e.g.,

```
REAL, DIMENSION(10) :: a=0.0, b=1.0, c
c = 5.0
```

Ex 6.2 Write a program to calculate

$$y = x^2 - 2x + 1$$

using array operations. First define an array for x which takes values from 0 to 5 with 0.25 intervals, then calculate y using array operations on x.

Output your results as a list of (x, y) pairs.

Ex 6.3 Consider a projectile fired from a launcher situated at the origin at an initial velocity of 25 m s^{-1} at an angle of $40°$. Using array operations, calculate the position of the projectile (neglecting any resistance) for a time interval of 0 to 3.5 seconds with a 0.1 second interval.

Output your results as a list of (t, x, y) triplets.

Can you write the program so it is easy to change parameters such as the time-range, the interval, the initial velocity and the initial angle?

Ex 6.4 Write a program to read in a set of real numbers (the number of elements to be specified by the user, but with a maximum of 20) and output the average of the values. Hint, you will probably need to address each element in the array individually within a block `DO` loop.

6.3 Intrinsic Functions for Arrays

6.3.1 Previously Encountered Intrinsic Functions

Most of the intrinsic functions that we saw for scalar values in section 3.6 also work for arrays. As with the intrinsic operators, they act on the array element by element as if the function was in a block `DO` loop. For example

```
REAL, DIMENSION(30) :: a, b, c

a = SIN(b)      ! = (/ SIN(b(1)), SIN(b(2)), ... /)
a = EXP(c)      ! = (/ EXP(c(1)), EXP(c(2)), ... /)
a = MAX(b,c)    ! = (/ MAX(b(1),c(1)), MAX(b(2),c(2)), ... /)
```

Ex 6.5 Consider the projectile in exercise 6.3 again. Rearrange the equations that you derived to calculate the time (when $t > 0$) that the projectile lands (i.e., $y = 0$), and the distance that it has travelled in the x-direction.

Using array operations, calculate the distance travelled in the x-direction for initial angles in the range of 0° to 90° at 1° intervals.

Output your results as a list of (θ, t, x) triplets.

At what angle does the projectile travel the furthest? Is this what you would have expected?

6.3.2 Array Specific Intrinsic Functions

As well as these functions, Fortran 90 has a set of intrinsic functions designed especially for arrays. These functions might perform array calculations (such as the sum of the elements), they might provide information about the data (such as the location of the maximum value) or they might give information about the array itself (such as the number of elements in the array). We will look at these functions in a bit more detail than we looked at scalar intrinsic functions in section 3.6.

First of all we need to look at the idea of a `mask`. This is an array (which is conformable with any other array argument of the function) of logical data (i.e., `.TRUE.` or `.FALSE.`). The value of the function will depend somehow on this mask. For example, the `ALL` function will return true if all values of `mask` are true, so

```
LOGICAL, DIMENSION(3) :: mask1, mask2
LOGICAL :: l1, l2

mask1 = (/ .TRUE., .TRUE., .TRUE. /)
mask2 = (/ .TRUE., .FALSE., .TRUE. /)

l1 = ALL(mask1)     ! = .TRUE.
l2 = ALL(mask2)     ! = .FALSE.
```

This becomes more useful when we use relational operators with arrays, for example,

```
  INTEGER, DIMENSION(3) :: a
  LOGICAL, DIMENSION(3) :: mask1, mask2
  LOGICAL :: l1, l2

  a = (/ 1, 3, 4 /)
  mask1 = a > 0       ! = (/ .TRUE., .TRUE., .TRUE. /)
  mask2 = a > 2       ! = (/ .FALSE., .TRUE., .TRUE. /)

  l1 = ALL(mask1)     ! = .TRUE.
  l2 = ALL(mask2)     ! = .FALSE.

! or perhaps more conveniently written as

  l1 = ALL(a>0)       ! = .TRUE.
  l2 = ALL(a>2)       ! = .FALSE.

! likely to be useful in a block IF construct

  IF (ALL(a>0)) THEN
    ...
  END IF
```

The mask array may also be used optionally in some functions (this will be denoted in italics, e.g., *mask*). In this context it will be used with an array, `arr`, which it conforms with and the function will operate only on the elements of `arr` where the corresponding elements of *mask* are true.

For example, the `MINVAL` function returns the minimum value of an array, when it is used with a *mask* it returns the minimum value of the array when the corresponding values of *mask* are true, e.g.,

```
  REAL, DIMENSION(3) :: a
  LOGICAL, DIMENSION(3) :: mask
  REAL :: min1, min2, min3

  a = (/ 3.2, 1.4, 2.7 /)
  mask = (/ .TRUE., .FALSE., .TRUE. /)

  min1 = MINVAL(a)          ! = 1.4
  min2 = MINVAL(a, mask)    ! = 2.7
  min3 = MINVAL(a, a>2.0)   ! = 2.7
```

A list of some of the more useful functions can be found in table 4

Ex 6.6 Modify your program from exercise 6.4 where you calculated the average of a set of values so that it uses the `SUM` function instead of a block `DO`. Add a section that calculates the average of the positive data only using a `mask` in the `SUM` function (you may also need to use the `COUNT` function).

Function	Purpose
`ALL(mask)`	Returns `.TRUE.` if all elements of mask are `.TRUE.`
`ANY(mask)`	Returns `.TRUE.` if any elements of mask are `.TRUE.`
`COUNT(mask)`	Returns the number (integer) of elements of mask that are `.TRUE.`
`MAXLOC(arr, mask)`	Returns the array index (integer) of the maximum value of `arr`
`MAXVAL(arr, mask)`	Returns the maximum value of the elements in `arr`
`MERGE(arr1, arr2, mask)`	Returns an array where elements take their value from `arr1` if the corresponding element of `mask` is `.TRUE.`, otherwise it takes the element from `arr2`
`MINLOC(arr, mask)`	Returns the array index (integer) of the minimum value of `arr`
`MINVAL(arr, mask)`	Returns the minimum value of the elements in `arr`
`PRODUCT(arr, mask)`	Returns the product of all the elements in the array
`SIZE(arr)`	Returns the number (integer) of elements in the array
`SUM(arr, mask)`	Returns the sum of all the elements in the array

Table 4: Table of some array intrinsic functions.

6.4 Multi-Dimensional Arrays

6.4.1 Defining Multi-Dimensional Arrays

So far we have used only one-dimensional arrays, Fortran 90 also has facilities for defining and working with multi-dimensional arrays (e.g., 2D, 3D, etc). The maximum allowed is a seven-dimensional array.

You can declare a higher-dimensional array as follows,

```
INTEGER, DIMENSION(5,3) :: a    ! 2D arrays
REAL, DIMENSION(10,4) :: b
INTEGER, DIMENSION(6,3,5) :: c  ! 3D arrays
REAL, DIMENSION(3,10,30) :: d
REAL, DIMENSION(3,4,6,2,10,10)  ! 6D array
```

This form of the `DIMENSION` statement declares the number of elements in each direction. So in two dimensions, the statement `DIMENSION(`m`,`n`)` declares an array which has m elements in the first direction and n elements in the second, as illustrated below.

`arr(1,1)`	`arr(2,1)`	...	`arr(`m`,1)`
`arr(1,2)`	`arr(2,2)`	...	`arr(`m`,2)`
⋮	⋮	⋱	⋮
`arr(1,`n`)`	`arr(2,`n`)`	...	`arr(`m`,`n`)`

This means that the array has a total of $m \times n$ elements.

This brings up an important point about memory allocation. On a typical 32 bit machine (which is what you will generally be using), the computer allocates 32 bits (or 4 bytes) to store an `INTEGER` or a `REAL` variable. To store an array, the program will need to allocate memory for the number of elements times 4 bytes. Consider the following cases;

Case	No of elements	Memory Required
1D	100	400 B
2D	100×100	39 KB
	500×500	0.95 MB
3D	$100 \times 100 \times 100$	3.8 MB
	$250 \times 250 \times 250$	60 MB
4D	$100 \times 100 \times 100 \times 100$	381 MB
5D	$100 \times 100 \times 100 \times 100 \times 100$	37 GB
6D	$100 \times 100 \times 100 \times 100 \times 100 \times 100$	3.6 TB

As you can see, as you increase the dimension of the array or the number of elements in a multi-dimensional array, the amount of memory required increases dramatically. To run a code with a large array you will need this amount of memory available (plus whatever other memory resources you require) otherwise your program will crash (it will likely compile okay, but will crash when you try to run it - probably with a segmentation error). Most computers at the moment are unlikely to have more than 1-2 GB, many computers in the University may only have 64-128 MB of total memory. To create larger arrays, people often need to use supercomputers.

When using large or multi-dimensional arrays, always calculate the amount of memory required and check that it is within the amount of memory available on the computer. If not, you may need to use smaller arrays, or come up with a more clever way of doing things.

As with one-dimensional arrays, you may specify the lower and upper bounds for each (or any) dimension, for example;

```
INTEGER, DIMENSION(-10:10, -8:4) :: a
REAL, DIMENSION(3, 0:11, 0:100) :: x
```

6.4.2 The Rank and Shape of an Array

There are five technical terms that relate to multi-dimensional arrays, these are;

- Fortran arrays can have up to seven subscripts, each of which relates to one *dimension* of the array.
- The total number of dimensions of an array is known as the *rank* of an array (e.g., a 3D array has rank equal to 3).
- The *extent* of a dimension is the number of elements in that dimension of the array (e.g., a 2D array of 50×100 has an extent of 50 in the first dimension and 100 in the second dimension).
- The *size* of an array is the total number of elements which make up the array.
- The *shape* of an array is determined by it's rank and the extent of each dimension. It can be characterised by a rank 1 array with each element corresponding to the extent of each dimension (e.g., a 3D array of $3 \times 20 \times 50$ has a shape of `(/ 3, 20, 50 /)`).

6.4.3 Multi-Dimensional Array Constants and Initial Values

Multi-dimensional arrays may be referred to in a similar way to one-dimensional arrays. To address array elements independently you can type;

```
INTEGER, DIMENSION(5,5) :: arr
```

```
arr(1,1) = 1
arr(2,1) = 2
.
.
.
arr(4,5) = 9
arr(5,5) = 10
```

or perhaps more conveniently as

```
INTEGER, DIMENSION(5,5) :: arr
INTEGER :: i, j

DO j=1,5
  DO i=1,5
    arr(i,j) = i+j
  END DO
END DO
```

You can also write array constants for multi-dimensional arrays. To do this you write the array as a one-dimensional array and change its shape using the RESHAPE statement, e.g.,

```
REAL, DIMENSION(3,2) :: x

x = RESHAPE( (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /), (/ 3, 2 /) )
```

which gives the array

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{bmatrix}$$

So the RESHAPE statement takes the form

```
RESHAPE( arr_1d, newshape )
```

where *arr_1d* is a 1D array with enough elements to form the new array and *newshape* is the shape of the new array.

For larger arrays, it may make clearer code if you use continuation lines. For example;

```
REAL, DIMENSION(5,4) :: x

x = RESHAPE( (/ 1.0, 2.0, 3.0, 4.0, 5.0,    &
           &    2.0, 3.0, 4.0, 5.0, 6.0     &
           &    3.0, 4.0, 5.0, 6.0, 7.0     &
           &    4.0, 5.0, 6.0, 7.0, 8.0 /), &
           & (/ 5, 4 /) )
```

You may also use the implied DO construct in the multi-dimensional case when you create an array to be reshaped, e.g.,

```
REAL, DIMENSION(10,10) :: x

x = RESHAPE( (/ ((i+j, i=1,10), j=1,10) /), (/ 10, 10 /) )
```

6.4.4 Using Multi-Dimensional Arrays

Using arrays in expressions is exactly the same as in the one-dimensional array case. Array operations may only be carried out on arrays that conform, and while in the one-dimensional case that meant that they were of the same size, in the multi-dimensional case they must also be of the same shape.

So for example, a 10×20 array conforms with either another 10×20 array or a scalar. So

```
REAL, DIMENSION(4,30) :: a, b, c
INTEGER :: i,j
.
.
.
a = b*c

! or
DO j=1,30
  DO i=1,4
    a(i,j) = b(i,j)*c(i,j)
  END DO
END DO
```

This means we can initialise multi-dimensional arrays with the short cut

```
REAL, DIMENSION(5,20) :: a=1.0, b=0.0
```

Ex 6.7 Write a program to calculate $f(x, y) = \sin x \times \cos y$ for $-\frac{\pi}{2} \leq x \leq \frac{\pi}{2}$ and $-\frac{\pi}{2} \leq y \leq \frac{\pi}{2}$ with increments of $\frac{\pi}{10}$ in each direction.

Define `x` and `y` to be 11×11 arrays with values correctly set for the corresponding position on the grid, i.e.,

$$x = \begin{bmatrix} -\frac{\pi}{2} & -\frac{4\pi}{5} & \cdots & \frac{4\pi}{5} & \frac{\pi}{2} \\ -\frac{\pi}{2} & -\frac{4\pi}{5} & \cdots & \frac{4\pi}{5} & \frac{\pi}{2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -\frac{\pi}{2} & -\frac{4\pi}{5} & \cdots & \frac{4\pi}{5} & \frac{\pi}{2} \\ -\frac{\pi}{2} & -\frac{4\pi}{5} & \cdots & \frac{4\pi}{5} & \frac{\pi}{2} \end{bmatrix}$$

and

$$y = \begin{bmatrix} -\frac{\pi}{2} & -\frac{\pi}{2} & \cdots & -\frac{\pi}{2} & -\frac{\pi}{2} \\ -\frac{4\pi}{5} & -\frac{4\pi}{5} & \cdots & -\frac{4\pi}{5} & -\frac{4\pi}{5} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{4\pi}{5} & \frac{4\pi}{5} & \cdots & \frac{4\pi}{5} & \frac{4\pi}{5} \\ \frac{\pi}{2} & \frac{\pi}{2} & \cdots & \frac{\pi}{2} & \frac{\pi}{2} \end{bmatrix}$$

and use array operations to calculate $f(x, y)$.

Output your results.

6.4.5 Intrinsic Functions for Multi-Dimensional Arrays

All the mathematical and intrinsic functions that we have previously met for arrays also work for multi-dimensional arrays, however, there are a few additional functions that are specific to multi-dimensional arrays.

There is another option that can be used with array specific functions and that is to specify which dimension we wish to act in. For example, suppose we had a two-dimensional array of REAL numbers and wanted to find the sum of each row (or perhaps each column). We could do this as follows;

```
  REAL, DIMENSION(5,4) :: arr
  REAL, DIMENSION(4) :: tot_rows
  REAL, DIMENSION(5) :: tot_cols
  REAL :: tot_all

  arr = RESHAPE( (/ 4.2, 7.4, 1.8, 5.6, 3.7, &
                  & 9.3, 2.2, 6.4, 7.1, 8.3, &
                  & 1.7, 5.4, 8.6, 0.4, 6.9, &
                  & 6.2, 9.5, 2.4, 4.9, 1.3 /), (/ 5, 4 /) )

  tot_rows = SUM(arr, 1)
!          = (/ 22.7, 33.3, 23.0, 24.3 /)

  tot_cols = SUM(arr, 2)
!          = (/ 21.4, 24.5, 19.2, 18.0, 20.2 /)

  tol_all = SUM(arr)    ! = 103.3
```

In this case we have used the SUM command with an optional dimension argument. When we specified `SUM(arr, 1)`, Fortran summed the elements in the direction of the first dimension giving us an array of totals of each row. When we specified `SUM(arr, 2)`, Fortran summed the elements in the direction of the second dimension giving us an array of totals of each column. When we omitted the optional dimension argument then Fortran summed all the elements of the array giving us a grand total.

Similarly, you could find the minimum value in each row/column, the number of elements greater than 5 in each column and so on, e.g.,

```
  min_rows = MINVAL(arr, 1)
!          = (/ 1.8, 2.2, 0.4, 1.3 /)

  gt5_cols = COUNT(arr > 5.0, 2)
!          = (/ 2, 3, 2, 2, 2 /)
```

Table 5 shows some useful functions for multi-dimensional arrays. Where the dimension can be selected, the optional *dim* argument will be shown, e.g., `SUM(arr, mask, dim)`. Note that the optional *mask* argument works as before and must conform with `arr`.

Ex 6.8 Write a program (or modify your program from exercise 6.4 or 6.6) to read in five sets of real numbers (the number of elements in each set to be specified by the user, but to have a maximum of 20) and output the average and standard deviation of the each set. Store the data in a 2D array and use the SUM function with the optional *dim* argument.

Note, the standard deviation of a set of data, x_i, with mean, $\bar{x}$ is given by;

$$\sigma = \sqrt{\left(\sum_i x_i^2\right) - \bar{x}^2}$$

Function	Purpose
`ALL(mask, dim)`	Returns `.TRUE.` if all elements of mask are `.TRUE.`
`ANY(mask, dim)`	Returns `.TRUE.` if any elements of mask are `.TRUE.`
`COUNT(mask, dim)`	Returns the number (integer) of elements of mask that are `.TRUE.`
`MAXLOC(arr, mask, dim)`	Returns the array index (integer) of the maximum value of `arr`
`MAXVAL(arr, mask, dim)`	Returns the maximum value of the elements in `arr`
`MERGE(arr1, arr2, mask)`	Returns an array where elements take their value from `arr1` if the corresponding element of `mask` is `.TRUE.`, otherwise it takes the element from `arr2`
`MINLOC(arr, mask, dim)`	Returns the array index (integer) of the minimum value of `arr`
`MINVAL(arr, mask, dim)`	Returns the minimum value of the elements in `arr`
`PRODUCT(arr, mask, dim)`	Returns the product of all the elements in the array
`RESHAPE(arr, shape)`	Returns an array which is made up of the elements of `arr` but has a new shape as defined by the rank 1 array `shape`
`SIZE(arr, dim)`	Returns the number (integer) of elements in the array
`SHAPE(arr)`	Returns a 1D array containing the shape of `arr`, the number of elements in the new array is equal to the rank of `arr`
`SPREAD(arr, dim, ncopies)`	Creates a new array which is 1 rank higher than `arr` by copying `ncopies` of `arr` in the dimension specified by `dim`. Like making a book from copies of a single page
`SUM(arr, mask, dim)`	Returns the sum of all the elements in the array
`TRANSPOSE(arr)`	Transposes a rank 2 array so that the `(i,j)` component of the new array is equal to `arr(j,i)`

Table 5: Table of some multi-dimensional array intrinsic functions.

6.5 Flexible Array Processing

There are a couple of additional useful array features in Fortran that will be covered in this course. The first is the `WHERE` construct which allows us to perform array operations on selected elements of an array and perhaps a different operation on other elements.

The second is the ability to act on subsections of an array rather than the whole array or individual elements.

6.5.1 Masked Array Assignment

The `WHERE` construct allows us to perform array expressions on only certain elements of an array according to a `mask` or a `mask_expression`. In its simplest form it can be used as follows;

```
WHERE (mask_expression) array_assignment_expression
```

where the *`mask_expression`* conforms with the arrays in the *`array_assignment_expression`*. For example

```
REAL, DIMENSION(20) :: a, b
.
.
.
WHERE (a > 0) b = SQRT(a)
```

In this case, the square root of elements of `a` are assigned to the corresponding elements of `b` only where `a > 0` (as square roots of negative numbers are complex). You might achieve the same effect with a

block DO loop and a block IF construct as follows;

```
REAL, DIMENSION(20) :: a, b
INTEGER :: i
.
.
.
DO i=1,20
  IF (a(i) > 0) THEN
    b(i) = SQRT(a(i))
  END IF
END DO
```

In this case you might also want to set the other elements of b to something as well, and for this the block WHERE construct comes in useful. The syntax is,

```
WHERE (mask_expression)
  array_assignment_statements
ELSEWHERE
  array_assignment_statements
END WHERE
```

or just

```
WHERE (mask_expression)
  array_assignment_statements
END WHERE
```

And so in the previous example we might have

```
REAL, DIMENSION(20) :: a, b
.
.
.
WHERE (a > 0)
  b = SQRT(a)
ELSEWHERE
  b = -SQRT(-a)
END WHERE
```

Ex 6.9 Write a program to calculate the following function using a block WHERE construct.

$$f(x) = \begin{cases} -x^2, & \text{if } x < 0 \\ x^2, & \text{if } x \geq 0 \end{cases}$$

Calculate an array of x-values from minx=-1.0 to maxx=1.5 with the number of elements in the array being npoints=26. Make all of these values PARAMETERs. Then use the WHERE construct to calculate the function. Output the results as $(x, f(x))$ pairs.

Vary the values of minx, maxx and npoints.

6.5.2 Sub-Arrays and Array Sections

We have seen how to use whole arrays in expressions, however, sometimes we may only want to use a subsection of an array, perhaps the middle section of the array or just a line of a 2D array.

The simple form of addressing an array subsection is

```
arr(initial:final)
```

Where *initial* is the subscript of the first element of the subarray and *final* is the last element.

So if we have a 6 elements arrays we can set the middle 4 elements to 1 by

```
  INTEGER, DIMENSION(6) :: arr=0

  arr(2:5) = (/ 1, 1, 1, 1 /)
! so arr = (/ 0, 1, 1, 1, 1, 0 /)
```

The subsection `arr(2:5)` conforms with a rank 1 array with 4 elements.

The more general form of addressing an array subsection is

```
arr(initial:final:incr)
```

where *initial* and *final* are as before and *incr* is the step between element indices. All of these quantities must be `INTEGER`s.

So to set every other element and every third element in an array to 1 you might do something like

```
  INTEGER, DIMENSION(8) :: arr1=0, arr2

  arr1(2:8:2) = (/ 1, 1, 1, 1 /)
! so arr1 = (/ 0, 1, 0, 1, 0, 1, 0, 1 /)

  arr2(1:7:3) = (/ 1, 1, 1 /)
! so arr2 = (/ 1, 0, 0, 1, 0, 0, 1, 0 /)
```

There are some simpler forms of addressing subsections by excluding one or more of the arguments. If you omit *initial* then Fortran takes the subsection from the first element in the array. If you omit *final* then Fortran takes the subsection till the last element in the array. If you omit *incr* then Fortran assumes an step of 1.

So all of the following are shortcuts

```
arr(initial:final)
arr(initial:)
arr(initial::incr)
arr(:final)
arr(:final:incr)
arr(::incr)
arr(:)
```

Ex 6.10 Consider the following array declarations and subarray assignments. In each case work out what values the final array will have.

(a)
```
INTEGER, DIMENSION(10) :: arr1=0
arr1(:5:2) = (/ 1, 2, 3 /)
```

(b)
```
INTEGER, DIMENSION(10) :: arr2=0
arr2(::3) = 1
arr2(2::3) = 2
arr2(3::3) = 3
```

(c)
```
INTEGER, DIMENSION(10) :: arr3=0
```

```
arr3(1:3) = (/ 3, 2, 1 /)
arr3(4:) = arr1(4:) + arr2(4:)
```

(d)
```
INTEGER, DIMENSION(-4:4) :: arr4=0
arr4(:-1) = -1
arr4(1:) = 1
```

(e)
```
INTEGER, DIMENSION(-4:4) :: arr5=0
arr5(-4:4:2) = (/ 4, 2, 0, 2, 4 /)
```

(f)
```
INTEGER, DIMENSION(3,3) :: arr6=0
arr6(:, 1) = 1
arr6(1:3:2, 2:3) = 2
arr6(2, 2:3) = (/ 3, 4 /)
```

Program in each case to check your results.

As can be seen in exercise 6.10, we can mix and match subarrays and subscripts in multi-dimensional arrays. When doing this, we must take extra care that our subarrays conform. Consider the following;

```
REAL, DIMENSION(8,6) :: arr=0

arr(2:6, 3:5) = ...
```

Then in order to complete the expression, we need to use arrays or array subsections that conform and so we need to know the shape of the subsection. With multi-dimensional arrays the subscripts denote a section of the array in that dimension, so subsections in a 2D array will denote a rectangle within the main array, and in a 3D array it denotes a cuboid.

So `arr(2:6, 3:5)` has a shape of `(/ 5, 3 /)`, and to complete our example

```
REAL, DIMENSION(8,6) :: arr=0

arr(2:6, 3:5) = RESHAPE( (/ 1.0, 1.0, 1.0, 1.0, 1.0, &
                          & 2.0, 2.0, 2.0, 2.0, 2.0, &
                          & 3.0, 3.0, 3.0, 3.0, 3.0 /), (/ 5, 3 /) )
```

Which gives

$$\texttt{arr} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Ex 6.11 A function can be differentiated numerically using the formula

$$\frac{\delta f(x_i)}{\delta x} = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

Consider the function

$$f(x) = x\mathrm{e}^{-x}$$

Generate an array of x values from 0 to 1 with 11 points. Calculate the function above for each of these points using an array expression. Now calculate the numerical derivatives for each point using array subsections.

Differentiate the above function and in your program calculate the analytical derivatives for each point in your x array. Calculate the difference between the numerical and analytical derivatives.

Output your results as (x, f(x), df(x), analytic_df(x), difference) lines.

What happens to the the difference (and hence the numerical accuracy) when you vary the number of points in x?

7 Functions and Subroutines

7.1 Programming Units

So far in this course, we have only come across the notion of the PROGRAM unit. This is the block of commands that occur between the PROGRAM and END PROGRAM statements. In each program there must be exactly one program units. As we have seen, these look as follows:

```
PROGRAM progname

  ! specification statements

  ! execution statements

END PROGRAM progname
```

When writing programs, it would be useful to separate out parts of code that we wish to use over again (or perhaps in other programs). Consider a real life example, suppose we have a recipe for apple pie. It would be useful (and most recipe books do this) to have a separate recipe for pastry as this crops up in many recipes. Then in the apple pie recipe we can just say use the pastry recipe rather than include the extra instructions. This would also save space if we had many recipes that required pastry.

So, in Fortran 90 it would be useful to have a similar mechanism to define more general computer procedures. There are two types of procedures in Fortran 90, functions and subroutines. We can use these to split our programs up so that key parts of our program can be written as smaller self-contained subprograms.

7.2 Functions

We have seen already how intrinsic functions (such as SIN(x), EXP(x) and SQRT(x)) can be used to perform different calculations. When the program encounters one of these functions then it calculates the result and substitutes the answer into the main expression.

We can define our own functions. Instead of enclosing the specification statements in PROGRAM-END PROGRAM delimiters, we instead use the FUNCTION-END FUNCTION delimiters as follows:

```
FUNCTION myfunc(d1, d2, ...)

  ! specification statements

  ! execution statements

END FUNCTION myfunc
```

Here, *myfunc* is the name of the function and the dummy variables *d1*, *d2*, etc are used to denote any input to the function. There may be none, one or many of these depending on your requirements. The specification and execution statements are largely the same as you use in PROGRAM units.

We can best describe how to write functions by an example. In this example we will calculate cube roots by using logarithms. So

```
FUNCTION cube_root(x)

! again, always incude one of these
```

```
  IMPLICIT NONE

! you need to declare what type of output the
! function returns
  REAL :: cube_root

! now define the input variables
  REAL, INTENT(IN) :: x

! other internal variables
  REAL :: log_x

! do calculation
  log_x = LOG(x)
  cube_root = EXP(log_x/3.0)

END FUNCTION cube_root
```

7.2.1 The `FUNCTION` Unit

The `FUNCTION` unit acts in a similar way to the `PROGRAM` unit in that it contains the instructions for the computer to perform some task. `FUNCTION` units differ in that they are not the main body of the program (which is executed when you run the program from the command line), instead they define some sub-code that can be accessed and run by the main program (or other functions or subroutines).

The name of the function obeys all the usual naming rules that apply to program names and variable names. However, you should not give functions the same name as a variable in the `PROGRAM` (or other) unit that calls it.

Directly after the name of the function comes a pair of parentheses containing the names of dummy variables that are passed to the function. There may be none, one or many dummy variables. In the case that there are none, then the brackets must still be present.

In the above example, we have named the function `cube_root` (which is descriptive of what the function will do) and we have indicated that it will accept one input which we will store in the local variable `x`.

7.2.2 Specification Statements

As always, you should begin your specification statements with

```
  IMPLICIT NONE
```

You must declare the function as a variable, this tells Fortran what type of output the function will produce. For the example, we wish to output a `REAL` value (the cube root). The function take any of the variable types already discussed in this course and may also be array valued.

There is a shortcut here, you can declare the function line as follows

```
REAL FUNCTION cube_root(x)
.
.
.
```

```
END FUNCTION cube_root
```

This is okay in small programs, but in longer programs it can lead to code which is difficult to read. It is suggested that you do not use this shortcut.

The next step is to declare the dummy variables that are used for input. This introduces the `INTENT` statement. For functions, all dummy variable should have `INTENT(IN)`. This allows values to be passed from the main program to the function, but if the value of the variable is changed in the function, that change will not be passed back to the main program.

For our example, we have set the `REAL` variable `x` to have `INTENT(IN)`.

While these variables are used to pass information into the function, they are dummy variables local to the function and do not need to share the same variable names used when the function is called from the main program. Similarly, the main program may use variables with the same name as used in functions, Fortran will distinguish between the two and there will be no conflict.

Finally, we can declare other variables to be used in the function. These are local to the function, so variables with the same name can safely be used in the main program or other functions without any conflict. These are declared in the same way as you would declare variables in the main program block.

7.2.3 Execution Statements

The execution statements may include almost anything that you would put inside the main program. Expressions, block `IF` statements and block `DO` loops are all fine.

The special variable which has the same name of the function can be used to assign the final output of the function. So in the example, the special variable is `cube_root` and we assign the value that we want returned to the main program to this variable.

7.2.4 Calling the Function in a Program

Now we have written our function, we now want to know how to use it in our program. The general usage is similar to using intrinsic functions, except we must declare an external function first. This is done in the specification part of the program and is similar to declaring a variable, e.g.,

```
  REAL, EXTERNAL :: cube_root
```

So for our example, in the same file we would write

```
! main program
PROGRAM cube_root_test

  IMPLICIT NONE
! declare user functions
  REAL, EXTERNAL :: cube_root
! declare variables
  REAL :: x, y, cr1, cr2, cr3, cr4

! set some initial values
  x = 8.0
  y = 26.0

! calculate some cube roots
```

```
  cr1 = cube_root(125.0)
  cr2 = cube_root(x)
  cr3 = cube_root(y)
  cr4 = 3.0*cube_root(2.0*x)/2.0

  PRINT *, cr1, cr2, cr3, cr4

END PROGRAM cube_root_test


! cube root function
FUNCTION cube_root(x)

! again, always include one of these
  IMPLICIT NONE

! you need to declare what type of output the
! function returns
  REAL :: cube_root

! now define the input variables
  REAL, INTENT(IN) :: x

! other internal variables
  REAL :: log_x

! do calculation
  log_x = LOG(x)
  cube_root = EXP(log_x/3.0)

END FUNCTION cube_root
```

In this example, we can see that the function `cube_root` can be used as you would use an implicit function. It may form part of a larger expression.

There are a couple of fine points to notice. When we call `cube_root` in the main program, we can use numbers, variables and expressions as the argument. The input expression/variable will be first evaluated and then passed on the the function. Within the function, the dummy variable `x` will assume the value of the input. The dummy variable does not have to have the same name as the variable used as input in the main program.

Ex 7.1 Program in the `cube_root` program given above. Try changing the input values in the main program. Write a second function that calculates the fourth root.

Can you write a third function that calculates the nth root (hint, you will need to supply n as in input to the function as well)?

Test your new functions with some different values and output the results.

Ex 7.2 Write a function to calculate the factorial of an integer ($n! = n \times (n-1) \times ... \times 2 \times 1$). You may wish to use block `DO` loops.

Write a program to test you factorial function with different values. Output the results.

Ex 7.3 Combinations are used in probability to calculate the number of ways of combining things

are. For example, if you toss a coin twice, the is one way to get two heads (HH), two ways to get one head and one tails (HT and TH), and one way to get two tails (TT). Combinations are denoted C_r^n where n is the number of trials and r is the number of successes ($r \leq n$). They can be calculated by

$$^nC_r = \frac{n!}{r!(n-r)!}$$

Write a function to calculate combinations. You may use the factorial function that you wrote in exercise 7.2. Note, if both n and r are integers (≥ 0 and $r \leq n$ then the result of the combination is also an integer.

Write a main program to test your combination function and output the results.

Ex 7.4 Write a program to output the following

$$\begin{array}{ccccccc} {}^0C_0 & & & & & & \\ {}^1C_0 & {}^1C_1 & & & & & \\ {}^2C_0 & {}^2C_1 & {}^2C_2 & & & & \\ \vdots & \cdots & \cdots & \ddots & & & \\ {}^6C_0 & {}^6C_1 & {}^6C_2 & {}^6C_3 & {}^6C_4 & {}^6C_5 & {}^6C_6 \end{array}$$

You may use your combination and factorial functions from the previous exercises.

Can you name this structure of values?

In the previous three exercises we have also demonstrated the concept of modular development. In exercise 7.4 we want to print out the structure of combinations. In order to do this we need a function that calculates combinations which in turn needs a function that calculated factorials.

We first develop and test the factorial function in exercise 7.2. Once that works we develop and test the combinations function in exercise 7.3. Finally we write the main program to solve our problem.

When you are developing programs, first break the problem down into meaningful pieces which you can develop and test separately. This makes it easier to develop code and track down bugs in your code.

It also makes things simpler when developing code in a team. For example, one person could write the factorial function while another is writing the combinations function. The author of the combinations function does not need to know the details of the factorial function, they just need to agree on an interface in advance.

7.3 Subroutines

Subroutines work slightly differently to functions. The purpose of functions is to calculate some result and return that value to the main program. The purpose of a subroutine is to execute some code that doesn't necessarily produce a numerical result at the end. For example, you might use a subroutine to print something to the screen in a standard way, or write something to a file. You may also use subroutines to evaluate several different quantities and pass the results back to the main program using arguments which have `INTENT(OUT)` or `INTENT(INOUT)`.

Like the `FUNCTION` block, the subroutine block takes the following layout.

```
SUBROUTINE mysub(d1, d2, ...)

  ! specifications statements

  ! execution statements
```

```
END SUBROUTINE mysub
```

Consider the following example to calculate and output individual lines of a multiplication table.

```
SUBROUTINE mtable_line(a,b)

  IMPLICIT NONE

! we dont have to declare a type for the subroutine
! so just the dummy variable declarations
  INTEGER, INTENT(IN) :: a, b

! local variable declarations
  INTEGER :: ab

! execution statements
  ab=a*b
  PRINT *, a,' times ', b, ' = ', ab

END SUBROUTINE mtable_line
```

7.3.1 The `SUBROUTINE` Unit

The `SUBROUTINE` unit is very similar to the `FUNCTION` unit. You have the parentheses for the subroutines arguments and the subroutine name obeys all the usual rules.

In the above example, our subroutine was called `mtable_line` and it is called with two arguments, `a` and `b`.

A subroutine may have no arguments, in which case you should still include a pair of empty brackets after the subroutine name.

7.3.2 Specification Statements

The difference from the `FUNCTION` unit is that you don't have to specify what type of variable the function returns to the main program (or other procedure) as `SUBROUTINES` do not full fill this role.

You do need to declare what you dummy variables are though. There are more choices here than there are for `FUNCTIONS`. The `INTENT` statement can take the values `INTENT(IN)`, `INTENT(OUT)` and `INTENT(INOUT)`. The description of the three options are

1. `INTENT(IN)`, this means that the variable is used for input only. You can change the value of this variable without effecting it's value in the main program.

2. `INTENT(OUT)`, this means that the variable is used for output only. When the function is used in the main program, it will pass additional information out through this variable. The initial value of this variable may not be set from the main program.

3. `INTENT(INOUT)`, this means that the variable can be used for both input and output. The variable will be passed into the function and if it is modified during the function then the new value will be passed back.

For our example, we have set the INTEGER variables a and b to have INTENT(IN). However, we could modify our example so that it also outputs the product of a and b, e.g.,

```
SUBROUTINE mtable_line_mod(a, b, ab)

  IMPLICIT NONE

! we dont have to declare a type for the subroutine
! so just the dummy variable declarations
  INTEGER, INTENT(IN) :: a, b
  INTEGER, INTENT(OUT) :: ab

! execution statements
  ab=a*b
  PRINT *, a,' times ', b, ' = ', ab

END SUBROUTINE mtable_line_mod
```

7.3.3 Execution Statements

As before, pretty much most of the execution statements that can appear in PROGRAMs and FUNCTIONS can appear in SUBROUTINES.

7.3.4 Calling Subroutines in a Program

Subroutines are not used like functions and they do not return a value in that way. Instead they have to be explicitly called using the CALL statement, e.g.,

```
  CALL mysub(d1, d2, ...)
```

As subroutines do not have a variable type, then they do not need to be declares as external either. Fortran will assume this when it sees the CALL statement.

So for our example,

```
PROGRAM mtables

  IMPLICIT NONE
  INTEGER :: i, n

! select multiplication table
  n=7

! loop through to print each line
  DO i=1,12
    CALL mtable_line(i,n)
  END DO

END PROGRAM mtables


! subroutine to print a multiplication table line
```

```
SUBROUTINE mtable_line(a,b)

  IMPLICIT NONE

! we dont have to declare a type for the subroutine
! so just the dummy variable declarations
  INTEGER, INTENT(IN) :: a, b

! local variable declarations
  INTEGER :: ab

! execution statements
  ab=a*b
  PRINT *, a,' times ', b, ' = ', ab

END SUBROUTINE mtable_line
```

Ex 7.5 Write a subroutine that takes as an argument the radius of a circle and calculates and outputs the radius, diameter, circumference and area of the circle.

Test this subroutine for different radii.

Ex 7.6 Write a subroutine that will accept an array of 10 real numbers and calculate their mean and standard deviation

$$\begin{aligned} \bar{x} &= \frac{1}{N}\sum_i x \\ \sigma^2 &= \frac{1}{N-1}\sum_i (x-\bar{x})^2 \end{aligned}$$

Return the mean and standard deviation to the main program using a pair of dummy variables in the interface which have `INTENT(OUT)`.

Write a main program which has some values to test your function.

Ex 7.7 A simple way to sort an array of numerical values is to start with the first element then look at all the following elements in turn, if an element is smaller than the current first element, then swap the two numbers, otherwise move on to the next element. Then repeat this for the 2nd, 3rd, 4th, and so on elements.

So in pseudo code this would look something as follows:

```
xx = array of values to sort
loop though i=1,n-1
  loop through j=i+1,n
    if xx(j)<xx(i) then swap xx(j) and xx(i)
  end loop
end loop
```

Write this as a Fortran subroutine for a real array of 10 elements. Set `xx` to have `INTENT(INOUT)` so you can pass the result back via it's original variable.

Write a program to test you subroutine.

The `INTENT(INOUT)` option is often used for passing arrays to and from subroutines. As large arrays take up valuable memory, it is usually more convenient to reuse existing arrays rather than creating copies.

7.4 Arrays and Procedures

In exercises 7.6 and 7.7 we have seen how arrays can be incorporated as input and output variables in procedures. However, in both cases we fixed the size of the array to 10 elements in both the subroutine and the procedure. Ideally, the program and the subroutine should not need to know details of each others code. Also, it would be more convenient if we could write a subroutine (or a function) that can accept an array of arbitrary size. For example, it would be better to write a subroutine that would calculate the mean and standard deviation for an array with any amount of elements.

This can be done by also passing the number of elements in the array as an argument and specifying the array with `DIMENSION(`*n*`)` in the specification. For example,

```
SUBROUTINE print_sum_squares(n, xx)

  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  REAL, DIMENSION(n), INTENT(IN) :: xx

  INTEGER :: i
  REAL :: ssq

  ssq = 0.0
  DO i=1,n
    ssq = ssq + xx(i)**2.0
  END DO

  PRINT *, 'Sum of squares = ', ssq

END SUBROUTINE print_sum_squares
```

Ex 7.8 Modify your mean and standard deviation subroutine from exercise 7.6 so that it will accept an array of arbitrary size.

Write a program to test it with different size arrays.

Ex 7.9 Modify your sorting subroutine from exercise 7.7 so that it will accept an array of arbitrary size.

Write a program to test it with different size arrays.

You can do this for higher dimensional arrays. For example, to pass a two-dimensional array you would type something like

```
SUBROUTINE mysub(nx, ny, arr)
  INTEGER, INTENT(IN) :: nx, ny
  INTEGER, DIMENSION(nx,ny), INTENT(INOUT) :: arr
```

and so on for higher dimensional arrays.

There are a variety of other ways of using arrays of varying size in subroutines and functions. This include using arrays with different lower and upper bounds and passing the lower and upper bounds as

extra arguments in the procedure. These are beyond the scope of this course and are covered in more detail in Ellis, Philips and Lahey.

7.5 Array Valued Functions

As well as `REAL` and `INTEGER` valued functions, you may also have array valued functions. These might be declared as follows

```
FUNCTION myfunc(...)

  IMPLICIT NONE
  REAL, DIMENSION(10) :: myfunc
  .
  .
  .

END FUNCTION myfunc
```

You may not want the size of the returned array to have a fixed size, rather one specified on input, e.g.,

```
FUNCTION myfunc(n)

  IMPLICIT NONE
  REAL, INTENT(IN) :: n
  REAL, DIMENSION(n) :: myfunc
  .
  .
  .

END FUNCTION myfunc
```

There are some other technical specifics with array valued functions which are not covered in detail here, for further details, please refer to Ellis, Philips and Lahey.

8 File Handling

So far we have restricted ourselves to reading data from the keyboard and writing to the screen. For larger amounts of data, this becomes impractical and we want to be able to input data from a file (perhaps generated by an experiment or another computer program) and write it out to another file (perhaps so we can read it into MathCAD and plot it).

In Fortran, we can read and write files in a variety of formats and compressions, however, for this course, we will restrict ourselves to plain text files that we can read/write using a text editor.

8.1 Opening and Closing a File

A file can be opened using the `OPEN` statement. The open statement takes a list of arguments, some of which are mandatory, others are optional. As a minimum, you should use the following arguments:

```
OPEN(UNIT=n, FILE=filename)
```

Where *n* is a unique integer constant or variable to identify that is used to reference the file within your program and *filename* is a character constant or variable which refers to the name of your file. So for example

```
OPEN(UNIT=10, FILE="myfile.dat")
```

The file should be located in, or will be written to, the directory in which you run your program.

For the purpose of this course, you should also include the options `FORM="FORMATTED"`. The options here are `FORMATTED` or `UNFORMATTED`. However, in this course we will only deal with `FORMATTED` files. So for example

```
OPEN(UNIT=10, FILE="myfile.dat", FORM="FORMATTED")
```

The next option is to tell Fortran what your intention for the file is, will you be reading it or writing to it? This is the `ACTION` option. This may take the values `ACTION="READ"` and `ACTION="WRITE"`. This will restrict what you can do to a file. There is also a `READWRITE` option that allows you to read and write to the same file, however, this is dangerous and we will not use this option in this course.

The final option that we will look at (though there are other options available) is the `STATUS` option. This may take the values `OLD`, `NEW`, `REPLACE`, `SCRATCH` and `UNKNOWN`.

If the file status is `OLD`, then the file must already exist, whereas if the status is `NEW` then the file must not already exist and Fortran will create a new file with that name (and change its status to `OLD` so that subsequent attempts to open the file as `NEW` will fail).

If the file status is `REPLACE`, then Fortran will delete that file (if it already exists) and write a new file with the same name. If the file does not already exist, then it will act as if `NEW` were specified.

If the file status is `SCRATCH` then a temporary file will be created for the use of the program. This file will be deleted at the completion of the program. You do not specify a filename with this option.

The `UNKNOWN` option is implementation specific, and different Fortran compilers may do different things. The most common implementation is that if the file exists it acts as `OLD` and if it doesn't exist it acts as `NEW`.

A file may be closed using the `CLOSE(`*n*`)` statement, where *n* is the file reference. For example

```
OPEN(UNIT=20, FILE="myfile.dat", FORM="FORMATTED", &
    &  ACTION="READ", STATUS="OLD")
```

```
.
.
.
CLOSE(20)
```

8.2 Reading from a File

You can read data from an existing file using the READ statement in a similar way as you read data from the keyboard. The usage is slightly different from the form you have previously seen. It is

```
READ(UNIT=n, FMT=*) x, y, z
```

This will read in three quantities (of the type defined by x, y and z) from the file referenced by *n*. The usual rules for reading apply and you are not able to read in more data than exists in the file (the program will throw and end of file error).

Note, the READ *, x, y, z that you have seen so far is a special shortcut and means READ(UNIT=*, FMT=*) x, y, z. The * unit is a special unit that means the standard way that information is inputted or outputted (depending on context).

Ex 8.1 Create a file called test8.1.dat in emacs (or whatever) and type in the following:

```
1.0 2.0 3.0
4 5 6
```

Now type in and run the following program

```
PROGRAM iotest1

  IMPLICIT NONE
  INTEGER :: a, b, c
  REAL :: x, y, z

! open file
  OPEN(UNIT=10, FILE="test8.1.dat",  FORM="FORMATTED", &
      &  ACTION="READ", STATUS="OLD")

! read in data
  READ(UNIT=10, FMT=*) x, y, z
  READ(UNIT=10, FMT=*) a, b, c

! close file
  CLOSE(10)

! print out data
  PRINT *, 'x=',x
  PRINT *, 'y=',y
  PRINT *, 'z=',z
  PRINT *, 'a=',a
  PRINT *, 'b=',b
  PRINT *, 'c=',c

END PROGRAM iotest1
```

Try changing the numbers in the datafile and how much data is read in until you are happy you understand reading data from a file. Try creating a different data file with a different name and read data from that as well (you will want a different identifying number if you have more than one file open).

You can read in whole arrays in one go, for example, if you had a data file containing 100 numbers, you could read them all in as follows:

```
REAL, DIMENSION(100) :: arr
.
.
.
READ(UNIT=5, FMT=*) arr
```

Ex 8.2 Create a data file containing the following numbers.

```
2.5 4.1 0.9 8.3 1.7
6.5 3.5 7.9 3.1 0.2
```

Modify your sorting program from exercise 7.9 so that it reads in the data from the file and sorts those numbers.

Try changing the size of the array and the number of pieces of data in the data file.

8.3 Writing to a File

The method of writing files is very similar to this, except we need to change the value of some of the options in the `OPEN` statement and use the `WRITE` statement. The usage of the `WRITE` statement is similar to that of the `READ` statement, for example

```
WRITE(UNIT=n, FMT=*) x, y, z
```

This will write out three quantities (of the type and value defined by `x`, `y` and `z`) to the file referenced by *n*.

Ex 8.3 Type in and run the following program

```
PROGRAM iotest2

  IMPLICIT NONE
  INTEGER :: a, b, c
  REAL :: x, y, z

! initialise some data
  a=9
  b=8
  c=7
  x=0.1
  y=0.2
  z=0.3

! open file
  OPEN(UNIT=10, FILE="test8.3.dat",  FORM="FORMATTED", &
```

```
      &  ACTION="WRITE", STATUS="NEW")

! read in data
  WRITE(UNIT=10, FMT=*) x, y, z
  WRITE(UNIT=10, FMT=*) a, b, c

! close file
  CLOSE(10)

END PROGRAM iotest2
```

The program should have created the file `test8.3.dat` at the command line, look at the contents of this file by typing

```
dob@fortran:~> cat test8.3.dat
```

Run this program again. Did it crash? Do you know why it crashed?

Try replacing `STATUS="NEW"` with `STATUS="REPLACE"` and compile and run the program. Does it crash this time?

Try changing the values of the data written and the amount of data written. Instead of variables, try writing numerical and character constants as well. Do this until you are happy about writing files.

As with the `READ` statement, you can `WRITE` an array of data out to file in one go as follows

```
REAL, DIMENSION(100) :: arr
.
.
.
WRITE(UNIT=7, FMT=*) arr
```

Ex 8.4 Modify your sorting code in exercise 8.2 so that it writes the sorted numbers out to a new file.

Play with the code so that it works with different sizes of datasets.

Ex 8.5 Look back over some of the codes that you have written in the earlier chapters of this course. Modify them so that they input their data from a file and output the results to file.

Do this with as many programs as you wish until you are happy with reading and writing files.