An Introduction to Unix and Fortran Programming

Session 2003/2004

Contents

Prologue

This book has been written with the intention of introducing students to programming in Fortran within a linux computing environment.

The book takes a step-by-step approach to learning and includes many practical examples. To complement the book, there is a series of practical sessions that are designed to help with the understanding of the material in the book as well as in how to apply fortran programming to very realistic scientific problems.

This book has developed out of lecture notes from courses taught to undergraduates at the School of the Environment and to MSc students in the Computational Fluid Dynamics programme at the University of Leeds, UK.

Learning Objectives

The learning objectives for this module are as follows:

1. Basic understanding of the Unix computer operating system

2. Understanding of the use of a Unix text editor (vi editor)

3. Development of fundamental knowledge of the Fortran programming language

4. Development of programming and debugging ability in Fortran

5. Development of ability to apply Fortran to solve problems posed in a written or mathematical form

6. Analysis of data and interpretation of results

7. Time management associated with assigned coursework

1 Introduction

The focus of this lecture is on the operating system called Unix. Unix is used extensively by scientists and it is the operating system that will be used throughout this module. You have to know the basic commands of Unix in order to instruct the computer to do the work you require.

This lecture will also include an introduction to the editor called *vi*. *vi* allows you to create files containing information such as for writing letters or programs.

2 Basic Unix

2.1 Basic Unix commands

After you first log onto the computer you will be in what is called a directory and you will see a prompt. Prompts are often very different from one computer to another but that won't affect your use of the computer. The prompt maybe something like '`[dobbie@leeds1 dobbie]\$`' (some simple prompts are just '`$`'). When you first log in, you are in your home directory. You can check the directory you are in by typing *pwd*. The result may be something like

```
/nfs/env/people/dobbie
```

I would like to point out that commands that are built-in unix commands are in italic and commands that you type (which will be a mixture of unix commands and names that you decide on) are set off in quotations in these notes, '`like this`'.

To have a look at what is currently in your directory, type *ls*. This will show you what files or directories are in your home directory, if there are any at all. In the next section, you will learn how make files. I'm sure you can imagine that if you created lots of files on the computer, then the number of files that appear when you type '*ls*' could be overwhelming. You can get around this by making directories. To make a directory, type '*mkdir* `name`', where '`name`' is the name of the directory you want to make. You might want to make two directories, `programs` and `letters`. You can do this by typing '*mkdir* `programs`' and '*mkdir* `letters`'.

Now type '*ls*' and you will see these two names. To go into the programs directory, type '*cd* `programs`'. '*cd*' stands for change directory. If you write programs, you should write them whilst you are in this directory or move the files here after you write them. To go back to your home directory you can do it two ways, the first is '*cd* `..`', which moves you back up to your home directory. Another way is just by typing '*cd*', which takes you directly to your home directory no matter what directory you are in. You can make more directories inside of these directories if you like. Let's try it. From your home directory, type '*cd* `programs`' to get into the `programs` directory from your home directory. Now type '*mkdir* `mod2240`'. Now go into that directory by typing '*cd* `mod2240`'. You are now in directory `mod2240` that is in the directory called `programs` which was made in your home directory. To jump right back to your home directory, type '*cd*'. To go up a directory one at a time, from directory `mod2240`, type '*cd* `..`' and you will be in the `programs` directory. Type it again and you will be in your home directory. So '`..`' means the next directory on route to your home directory. (At least at this point, don't go up above your home directory. If you go there by mistake them just type '*cd*'.

The following are some basic unix commands that will allow you to use the unix system in a basic manner.

- *cd* – Change directory.
  Example, '*cd* `letters`'.
- *ls* – List the contents of the directory, files and directories.
- *clear* – Clear the screen.
- *cp* – Copy files. Copy them into another name or into a directory.
  Example 1, '*cp* `firstletter.txt letter1.txt`'. This just makes a new file (`letter1.txt`) with the same contents as `firstletter.txt`.
  Example 2, '*cp* `firstletter.txt programs`'. In this case, a new file is made in the directory named `programs`. The name of the file will be `firstletter.txt`. There will be two copies now, one in the current directory and one in the `programs` directory.
- *mv* – Move files. Similar to *cp*.
  Example1, '*mv* `firstletter.txt letter1.txt`'. This will rename the file as `letter1.txt` and delete the old file `firstletter.txt`. In contrast to this, *cp* keeps the old file too.
  Example 2, '*mv* `firstletter.txt programs`'. This will move the file to the `programs` directory and will not leave a copy in the current directory.
- *mkdir* – To make a new directory that is latched onto the directory you are currently in (current directory), type '*mkdir* `directoryname`'.
- *rm* – To delete a file, type '*rm* `filename`'. Be careful with this command. There are many options that can be used that could wipe out all of your work in all of your directories.
- *rmdir* – To remove a directory, type '*rmdir* `directoryname`'. Although files and directories look similar, you cannot remove a directory with just *rm*.

2.2 The *vi* text editor

If you want to make a file with a letter in it, then you need to be able to open a file and write. To do this, we will use the editor called *vi*.

The main thing to know about *vi* is that you are in either one of two modes. One is a writing mode (called insert mode) and the other is a command mode. Obviously, if you are in the writing (insert) mode, then this is when you write your letters or programs or whatever. When you are in the command mode, you are able to save the file and exit, or just exit and forget the additions to the file, etc. The command mode of *vi* allows you to do many more operations on the file than just save or quit the file. We will get into some of these in due course. I will soon describe how to get between the two modes as well.

2.2.1 Creating a file

We will begin by opening a file. We will begin a new file so you will have to think up a name. How about calling it '`proj1`' for the first project. Use regular alphabetical letters to start the name, but then you can use some special characters later in the word like a period, for example. Don't leave spaces when naming a file (Always make sure that the '`Caps Lock`' is not on). We will be writing a letter and so it will just be some text. It is helpful to indicate that we are using the file for a letter by appending a suitable (but optional) suffix to the name. So since it is going to be a text file, we perhaps should call the file '`proj1.txt`'. To start writing this letter in the file with this name, you type the following at the prompt: '*vi* `proj1.txt`'.

You will find that the screen changes and you have a square cursor at the top left and at the bottom left is the name of the file. When you first enter a file, you are in command mode. You need to get into insert mode in order to write letters or programs. To do this, type '`i`' or '`a`'. Try '`i`' first. Your screen will now indicate that you are in insert mode at the lower left corner. You are now ready to type a letter or a program, so start typing as normal for a letter. At any point, you can get back out to command mode by hitting the `Esc` button, which is normally at the top left of the keyboard. Hit `Esc` now and you will see the cursor backs up by one space to the last letter that you typed, unless the cursor is at the left-hand side. You are now ready to save the work. To issue these sorts of commands in the command mode, you need to type a colon followed by the command. To save the file, type '`:wq`'. The '`w`' indicates that you want to write (or save) this file to disk and the '`q`' indicates that you want to leave the file now. If you type '`:wq`' now, then you will end up back in your directory. Type *ls* and the name of the file you just created will appear, '`proj1.txt`'.

Let's go back into the file and add more text. Type '*vi* `proj1.txt`' and the file with the text will appear again. Again, you always start in command mode and your cursor is at the top left again. Use the arrows to get you to where you last wrote. Say the last words you wrote were '`I am working on this code`'. The cursor will go to the '`e`' of code but will not go any further. If you type '`i`' again, then you will run into trouble in that the cursor will push the last character to the right as you type more of the letter. Try it. Go to the last character in your letter and type '`i`' and type more in the letter. When you finish, you will find that the last letter when you first entered the file is pushed to the right. Get out of the file without saving by first getting to command mode (`Esc`) and then type '`:q!`'. The '`q`' is to quit and the '`!`' is to indicate that you don't want to save the new words that you added to the file. Go back into the file ('*vi* `proj1.txt`') and you will see the old file. Now use the cursors to get to the end of the last word that you wrote. Now use the other way to enter insert mode, type '`a`' for append. This will allow you to add text after the location of the cursor, whereas '`i`' is for insert at the cursor location.

What I have described above is the most difficult part to get used to at first. Once you can get into insert mode and back to command mode and add text effectively then you are well on your way to understanding the fundamentals of *vi*. Keep in mind that you can always hit '`Esc`' and go to command mode if you are unsure as to what mode you are in.

2.2.2 More on command mode

Once you have mastered the modes, then it is time to see what extra power the command mode has. Make sure your file has lots of writing in it, including at least five lines of writing and at least five words on each line. Say you just entered the file and you want to get down the page, then type '`control-d`'. This means hold the control key down and then press '`d`' whilst you are still holding the control key down. Try it. If you type '`G`' (Capitals), it will take you to the last line of the file. If you want to get back up to the top of the file then hit '`1 G`' (notice that there is a space between '`1`' and '`G`' so do them consecutively, releasing the '`1`' before hitting the '`G`'. Note that '`G`' is a capital and so it will be '`shift-g`' to get '`G`'. A dash is used to indicate that the two buttons are depressed at the same time. '`3 G`' will take you to the start of line `3`. To get to the end of the line, type '`$`'. To get to the start of the line, type '`^`'. Now try typing '`e`'. It will take you to the end of the first word. Do the same again and it will take you to the end of the next word. Type '`b`' and you will see the cursor move to the start of the word. Hit '`b`' again and it will take you to the start of the next word to the left or up a line if you are already at the left-hand side. At the start of a line, if you hit '`3 e`' then you will be taken to the end of the third word.

If you don't like the fourth line of work, then in command mode, hit '`4 G`' to take you to that line. Then type '`d d`' and the line will disappear. Consider that you change your mind and you wish that line were back again. Type '`u`' and it will reappear.

If you are on line two and you think there should be another line between line two and three, then in command mode, type '`o`' and the cursor will open a new line for you just below the position of the cursor at the time you hit '`o`'.

If you want to delete a word whilst you are in command mode, you have to place your cursor at the start of the word and type '`x`' repeatedly until the word disappears. Alternatively, if the word is four letters long then type '`4 x`' and it will go all at once.

If you delete a line or some text in the command mode by mistake, then type '`u`' and it will reappear again. Some configurations of *vi* allow you to continue to undo changes by repeated typing of '`u`'. Keep in mind that all changes between when you hit '`u`' and the last time you were in command mode will revert back to what they were or disappear, if they are new text. Experiment with '`u`' using text that you don't mind losing. When you master the command, it will allow you to avoid losses in the future. All of these commands are very useful, especially when you are dealing with computer programs.

The command mode has many more complicated functions and we will address some of them as the module progresses. Mastering an editor such as *vi* will be invaluable as a scientific programmer.

2.2.3 'Command mode' commands

'`:wq`'	save and exit
'`:w`'	save and stay in the editor
'`:q`'	quit a file that hasn't been altered, without saving
'`:q!`'	quit a file that has been altered, without saving changes
'`:w!  filename`'	save the contents of the file to a new filename named '`filename`' and stay in the file
'`i`'	insert mode
'`a`'	insert mode (insert after the position of the cursor) (append)
'`d d`'	delete a line (consecutive '`d`'s)
'`4 d d`'	delete four lines. Replace 4 with any number
'`e`'	go to the end of next word to right (also '`4e`', etc.)
'`b`'	go to the beginning of the word toward the left (also '`4b`', etc.)
'`u`'	undo the last change you made in insert mode (since you went into insert mode)
'`x`'	to delete characters
'`shift-j`'	to join lines together
'`control-d`'	jumps you down a large file in pages
'`control-u`'	jumps you up a large file in pages
'`G`'	takes you to the last line in the file
'`3 G`'	indicates jumping to line 3 (exchange 3 with any number, e.g. '`3432 G`')

3 Fortran Programming – Introduction

3.1 First Fortran program

In this section, you will learn how to write a very simple Fortran code, compile it, and execute it (run it).

In a previous section, you learned how to open and write words in a file (using *vi*). Writing a computer program is similar to this in that a program is simply a file with words in it. What makes it a program is that the words are instructions that Fortran can understand and that these instructions can be used to make the computer do specific operations. We will learn about some of the operations computers can do throughout this course.

We need to get more specific about some points. When you write a Fortran program in a file, this is called 'source code'. You will then run an operation called 'compiling and linking', which will make a new file. This new file is the compiled computer program or executable. The compiling and linking stage has converted your Fortran instructions into instructions for the computer, called machine language.

To write Fortran source code, you need to adhere to very specific rules as to where to put things and what to write in order for the program to work.

To make a Fortran source code, you need to first open a file to write in. You need to call the file a name that ends with '`.f`', e.g. '`program1.f`'.

Our first code will convert a temperature in centigrade, `TC`, into Fahrenheit, `TF`. Before we start understanding our first computer code, we need to learn about Fortran's line formats first.

3.1.1 Line formats

Fortran, because it dates back to times when computer programs were written on punch cards, it has a strict line format. You can only write on 80 columns on each line.

Columns 1 through 5

Columns 1 through 5 are used for statement labels (we'll get into that later). If you insert a '`c`' (or '`C`') or a '`*`' in column 1, then the line will be just treated as a comment line when you compile the program and it will be ignored when the compiler converts the source code into machine language.

Column 6

Sometimes, you will find that you don't have enough space to write all the commands that you want on one line within the restrictions imposed. If you place a '`+`' or an '`&`' in column 6 and this indicates that you wish to continue writing commands from the previous line onto the current line. It is called a continuation.

Columns 7 through 72

Fortran statements are to be written in columns 7 through 72. There is no need to left-justify the commands. This is where the vast majority of all your Fortran instructions will be placed.

Columns 73 through 80

These lines will not be used by Fortran and can be used to number the lines of the source code or for other purposes. Historically, the punch cards were labelled in these columns so that if you dropped a stack of the cards, then you would be able to easily reorder them.

First program

We are ready to analyse our first program source code. The source code is shown below in Figure 3.1. You should begin the first line with the name '`program`' followed by a label that will remind you what the program does. On the next line, I have written a statement 'TC=30.0'. I have just made up this variable 'TC' to mean temperature in Celsius. The second line of the source code defines the variable 'TC' to have a value of 30.0. When the program is compiled and run, after line 2, the 'TC' can be considered to be equal to 30.0 unless it is changed. The third line starts with a variable 'TF'. When the program is compiled and run, 'TF' will be assigned the Fahrenheit value of 30 degrees Celsius. The conversion to Fahrenheit is given by multiplying by 9.0, dividing by 5.0, and adding 32.0.

The fourth line in Figure 3.1 prints the two variables. When I say print, I don't mean to the printer, necessarily. Rarely do programs write directly to the printer from their computer program. In this case, the code is instructing the computer to output the values of 'TF' and 'Tc' to the screen. I will explain all of the symbols in the print statement at a later time. The fifth and sixth lines are the lines that all Fortran programs end with, `STOP` and `END`.

```
PROGRAM TMPCON
TC=30.0
TF=9.0*TC/5.0+32.0
PRINT *,TC,TF
STOP
END
```

Figure 3.1: Example 1

This is your first program. To get it ready for use, you need to compile and link it. Type '`f77 program1.f`' and this will create a new file called `a.out`. The compiling and linking converts the code into an `a.out` file which is in a language that the computer now understands, machine language. To run the program `a.out`, at the prompt simply type '`a.out`' and hit '`return`'. The code will run and before the prompt comes back, the result will appear something like '`30.0 86.0`'. Keep in mind that when you first write code you will often have small errors or mistakes in the source code. The compiler and linker will notify you of all the mistakes that it can find after you type '`f77 program1.f`'. If it finds any errors, then '`a.out`' will not appear in your directory and you need to go back and look over your file in `program1.f` to find the errors. The compiler (and linker) will give you hints as to what error you might have and it will tell you the line number in the source code that the error occurred. Sometimes the compiler will just issue warnings to bring something to your attention. These warning messages will not stop a compilation, but look at the message carefully to see if the warning is highlighting something that was unintended.

This has shown how easy it is to write and run a basic Fortran program. The last program is not in a good form, however. We will now add some things to improve the programming. I displayed the last code only to show you how easy it is to do something basic in Fortran. A better code would be written as in Figure 3.2.

```
      PROGRAM TMPCON
c     Written by Dr Steven Dobbie, Oct 1, 2002.
c     Computes the temp conversion from Celsius to Fahrenheit.
c     Version 2
      IMPLICIT NONE
      REAL TC,TF
c     Assign a value for Temperature in Celsius
      TC=30.0
c     Compute the value of Temp in Fahrenheit
      TF=9.0*TC/5.0+32.0
c     Output the results
      PRINT *,'Program: Temperature Converter'
      PRINT *,'Celsius T= ',TC,' Fahrenheit T= ',TF
      STOP
      END
```

Figure 3.2: Example 2

It is much better for many reasons. Lines 2 to 4 are comment lines that tell people looking at the code (such as yourself when you come back to it at a later date) who wrote it, when you wrote it, and what version it is and what it does.

Implicit none is stated on line 5. This is an instruction to the compiler that all variables (like `TC` and `TF`) should be defined. I have defined the variables on line 6 as both being real numbers. Variables that are to hold numbers can be defined as real or integer (also complex). When you write values for real numbers they should always have a decimal place, even if the number has nothing beyond the decimal place, e.g. 19.0.

3.1.2 Declarations

If you don't declare variables, like in Figure 3.1, then when Fortran sees a new variable it automatically sets it to be real if the variable begins with the letter '`A`' through '`H`' or '`O`' through '`Z`'. Variables '`I`' through '`N`' are considered to be integers. You see that in the first example, I made sure that I started the variable names with something that would make them automatically declared as real, '`T`' (as in '`TC`' and '`TF`'). It is much better practice to define implicit none and declare all variables as I did in the second example. It may take a bit more time, but it can help you if your program won't compile. If you declare a variable with the same name by mistake then it will be noted in the compiler stage as an error. Also, if you misspell a variable then the compiler will alert you to the undeclared variable (as long as your misspelling didn't result in a name of another variable! This would be a case when the compiler and linker wouldn't catch the error in the code. It isn't deemed to be an error by the compiler, but your program wouldn't work correctly.).

Back to the example

Line 7 (Figure 3.2) is a comment that tells you the next line is setting the Celsius temperature variable to a certain value. It may seem obvious and doesn't need stating. You will learn how many comments to put in your code to remind you of what it does without filling your source code with too many comments. Efficient use of variable names and comments help when you return to a program a year later or give a code to someone else to learn and use.

Lines 8 through 15 (Figure 3.2) are similar but with some extra comments. You will also notice that lines 12 and 13 give a much better format for the program output. You should try to include a very descriptive output that is nicely formatted so that it is easily understood.

I have stored this new program in a file called `program2.f`. (This is most easily done by ‘`cp program1.f program2.f`’ and then edit the changes into the new `program2.f`). Compile this source code by typing ‘`f77 program2.f`’ and run the resulting executable by typing ‘`a.out`’ at the prompt.

Explicit declaration

When you want to declare a variable, then you must adhere to some rules. Variable names can only be 6 characters long and must begin with an alphabetical character. In choosing variable names you can include numbers after the first alphabetical character, but don't begin a variable name with a number. Keep in mind that Fortran will not distinguish between upper and lower case variables, so if in your code you define ‘`REAL Tc,TC`’, then the compiler will complain that you have tried to declare the same variable twice! Standard Fortran will exclusively be in uppercase since in the past punchcards only allowed for the upper case. It is good practice to write all occurrences of a variable in the same way, i.e. ‘`TC`’ throughout the code is always ‘`TC`’ (never ‘`tc`’). You should write your programs in upper case only. Comments can be in lower case though.

There are some special characters in Fortran that shouldn't be used for variable names. These characters are ‘=’, ‘+’, ‘-’, ‘*’, ‘/’, ‘(’, ‘)’, ‘,’, ‘.’, ‘$’, ‘'’, ‘:’. These have special uses and you will learn many of these uses as you progress through the course.

When you declare a variable as real then it can hold a number of only a certain size. You must be aware in your program that the variable will not exceed this limit. The exact size that a real variable can be depends on the specifics of the computer, but an order of magnitude is roughly $\pm 10^{\pm 16}$ (plus and minus) with about seven significant digits carried. If you require a larger range in your calculations or if you believe that the number of significant digits is too small, then you can declare variables to be ‘`double real`’ (this may not work for Linux on a PC, so instead declare as ‘`real*8`’). Then, the range that the variable can take is roughly $\pm 10^{\pm 300}$ with about fourteen significant digits.

Fortran also allows for variables to store words. You define these variables are CHARACTERS. We will consider characters at a later date.

3.1.3 Mathematical operators and representation

Much of the code that I showed you in the examples of Figure 3.1 and 3.2 looks like algebra, such as 'TF = 9.0*TC/5.0+32.0'. It is important to note that this is more of an assignment rather than an algebraic equation. We are setting the variable on the left-hand side equal to the value of the right hand side (not the reverse). Until that line, 'TF' wasn't even defined to have a value.

Basic mathematical operations are defined in Fortran as '=', '+', '-', '*', '/', and '**' (exponentiation, e.g. 2**3=8). Scientific notation is represented in the following way.

Number	Scientific notation in Fortran
34567.34	3.456734E4 or 3.456734E+04

3.1.4 Reading input from the screen

We are going to return to Example 2 (Figure 3.2) to update the program in a small but significant way. You may have realised that if you want to use the program2.f to compute the temperature conversion from Celsius to Fahrenheit, then it would be cumbersome. This is because each and every time you have a different value for the temperature in Celsius, you would have to edit the program, compile and link it, and then run it. It might be easier to use a calculator. Consider the Example 3 below,

```
      PROGRAM TMPCON
c     Written by Dr Steven Dobbie, Oct 1, 2002.
c     Computes the temp conversion from Celsius to Fahrenheit.
c     Version 3
      IMPLICIT NONE
      REAL TC,TF
c     Assign a value for Temperature in Celsius
      PRINT *,'Please enter a temp in Celsius'
      READ (5,*) TC
c     Compute the value of Temp in Fahrenheit
      TF=9.0*TC/5.0+32.0
c     Output the results
      PRINT *,'Program: Temperature Converter'
      PRINT *,'Celsius T= ',TC,' Fahrenheit T= ',TF
      STOP
      END
```

Figure 3.3: Example 3

The line stating that TC=30.0 has been deleted and in place of it is a print statement which prints 'Please enter a temp in Celsius' and a read statement which reads the value entered into the variable TC. If you compile (and link) this source code and run it, you will be asked for a Celsius temperature and after you enter the value, it will print out the conversion like before. Except now, you just run the code again (type `a.out` at the prompt again) to enter a different value for the temperature in Celsius.

3.2 Real and integer calculations

Let us assume that all variables that we are dealing with have been declared integers. When we evaluate an expression using these variables we should not be surprised to find that the answer is still an integer! All well and good you may say, but we must be careful about how we approach this since the expression 9/5 will give the value 1 (remember that Fortran assumes numbers without a decimal point are integers) when we wish that the actual value be a real number.

More importantly, we must be careful in using mixed mode expressions, i.e. using real and integer variables together in the same expression. Take, for example, the well known formulae for converting centigrade to Fahrenheit:

```
Fahrenheit=9.0*centigrade/5.0+32.0
```

Okay, but what would happen if we wrote it like this:

```
Fahrenheit=9/5*centigrade*32.0
```

The effective answer would be to simply add 32 to the value of centigrade since 9/5 would result as a value of 1. Never write mixed mode expressions if you want to get it correct, always put in a decimal point. (Section 3.2 is quoted directly from Mech5510 course notes).

4 Fortran Programming – Decisions and Input/Output

4.1 Decisions in programs (`IF` statements)

In this section, you will learn how to include decisions in your Fortran program. Decisions are ubiquitous in computer programs. The more decisions you include in your program, the more situations your computer program can model. In this section, we will learn the basis of all decision coding in Fortran, the '`IF` statements'. The form of the '`IF` statement' is as follows:

```
IF (criterion) THEN
   executable statement
ELSE IF (criterion) THEN
   executable statement
ELSE IF (criterion) THEN
   executable statement
ELSE
   executable statement
ENDIF
```

When statements like the ones above are included in your program, when you run the program it will at the first line above come to a decision. If the above statements are included in your program, then you will be able to consider decisions. When the program is compiled, run, and arrives at the first line shown above, it will evaluate the criterion to determine if it is true or not. If it is true then the program will continue onto the next line and execute the statements on that line. If the criterion is not satisfied, then the running program will proceed to the next `ELSE IF` statement (3rd line). It will evaluate the criterion on the third line and determine if it is true or not and will then proceed to execute the 4th line of code or jump to the next `ELSE IF` statement (or `ENDIF`). When the running program reaches the `ENDIF` statement it proceeds to execute any code that might follow this '`IF` statement' block.

Keep in mind, that if any criterion is true and the code proceeds to execute the statements after that criterion, then the code will skip the next `ELSE IF` statements and proceed right to `ENDIF` and continue in the program.

Okay, let's look at a practical example. Say your program has been working with a variable that is called `T` for temperature. The program has considered all kinds of influences on the temperatures and you don't know what the current temperature is. Perhaps you are interested to know if the temperature is above or below freezing in centigrade (0 degrees). You can add this following bit of source code to evaluate the value of your temperature variable, `T`, at this point in your code.

```
IF (T .GT. 0.0) THEN
   PRINT *,'The temperature is currently above freezing'
ELSE IF (T .EQ. 0.0) THEN
   PRINT *,'The temperature is at the freezing level'
ELSE
   PRINT *,'The temperature is currently below freezing'
ENDIF
```

Figure 4.1: Example 4

The seven lines in Figure 4.1 will determine if the temperature is above, at, or below freezing and will print out the result to the screen.

4.1.1 Indentation

Indentation is extremely important in writing your program. You should adhere to the form that is presented in this course so that you and others will be able to see clearly the form of the Fortran statements. In Example 4, you will notice that the executable statements (print statements) are indented relative to the '`IF` Statements'. Everyone who knows Fortran will expect your code writing to adhere to this form. There are similar indentations for other operations, such as loops. I will highlight indentation throughout this module.

4.1.2 Criterion testing

You can see in Example 4 that the first criterion testing is 'T .GT. 0.0'. This is a statement that tests if T is Greater Than (.GT.) zero. If it is true, then the program will execute the statement on the next line which is a print statement saying that the temperature is greater than zero. After this, the program will proceed to the ENDIF statement. If 'T .GT. 0.0' is not true, then the program (when running) will proceed to the next 'IF statement' in the block, which contains the criterion 'T .EQ. 0.0'. If the temperature is EQual to (.EQ.) zero, then it will execute the statement (or statements) on the following lines and up until the next 'IF statement' in the block. It will print to the screen the statement that the temperature is currently zero and then proceed to the ENDIF in this case.

Keep in mind that you don't need all of the statements to be present. When the code is executing, you might want to only know if the temperature, T, goes above a realistic value, say 60 centigrade. In this case, you will be alerted to your code not running correctly if you expect your code should never attain such values. To test for this error, you might include a statement like in Figure 4.2.

```
IF (T .GT. 60.0) THEN
   PRINT *,'Error: Temperature has exceeded a realistic value'
ENDIF
```

Figure 4.2: Example 5

The above print statement will only be used if the criterion 'T .GT. 60.0' is true. It is unrealistic to print out every variable that your code uses, but if you have physical justification of why a value shouldn't exceed a certain value in your code then it is useful to include a print statement such as the one above to notify you if does.

You could test to see if your model exceeds a lower limit too by including the following. In addition to the upper temperature limit, say you don't physically expect the temperature variable to go below -70 centigrade, then you could test for both of these errors as in Figure 4.3.

```
IF (T  .GT. 60.0) THEN
   PRINT *, 'Error: Temperature has gone above 60.0 Celsius'
ELSE IF (T .LT. -70.0) THEN
   PRINT *,'Error: Temperature has gone below -70.0 Celsius'
ENDIF
```

Figure 4.3: Example 5

The following are the standard criterion testing conditions.

Relational operation	Test Performed
`A .GT. B`	$A > B$?
`A .GE. B`	$A \geqslant B$?
`A .EQ. B`	$A = B$?
`A .LT. B`	$A < B$?
`A .LE. B`	$A \leqslant B$?
`A .NE. B`	A not equal to B?

You can combine tests into fewer '`IF` statements' by using the '`AND`' or '`OR`' statements. Let's re-look at Example 5. This can be written also as in Figure 4.4.

```
IF ( (T .GT. 60.0)  .OR. (T .LT. -70.0) ) THEN
   PRINT *, 'Error: Temperature is out of bounds, the temp is ', T
ENDIF
```

Figure 4.4: Example 6

The general form of Example 6 is

```
IF (  (criterion) .OR. (criterion)  ) THEN
```

Take careful note of the brackets used.

Let us consider a more complex situation. Your model is simulating climate, and some of the variables contained in your program are surface temperature, `ST`, latitude, `LAT`, pressure, `P`, etc. I want to focus on the temperature and latitude variables. You may deem that for latitudes near the equator (e.g. plus or minus 30.0 degrees) that you don't expect the surface temperature to go below −30.0 degrees Centigrade (it could easily go lower in the high latitudes). So your program is analysing at various latitudes and storing the surface temperature each time in the variable `ST`. You can then test this variable to see if it is taking on realistic values in the low latitudes by the statement in Figure 4.5.

```
IF ( ( (LAT .GE. -30.0) .AND. (LAT .LE. 30.0) )  .AND. (ST .LT -30.0) ) THEN
   PRINT *,'Error: Equatorial temperature is too low ',ST
ENDIF
```

Figure 4.5: Example 7

Example 7 takes on the following general form

```
IF (  ((criterion) .AND. (criterion)) .AND. (criterion)  ) THEN
```

Take careful note of the use of brackets to separate the criterion being tested.

4.1.3 Nesting 'IF statements'

Example 7 shows a complicated testing that may not be easy to see for someone else reading your code (or you returning to it at a later date). You can 'nest' statements to accomplish more difficult tasks but still retaining clarity. It is important to note the indentation that I have used in Figure 4.6.

```
IF (LAT  .GE.  -30.0) THEN
   IF (LAT .LE. 30.0) THEN
      IF (ST .LT. -30.0) THEN
         PRINT *,'Error: Equatorial temperature is too low, ST= ',ST
        ENDIF
     ENDIF
ENDIF
```

Figure 4.6: Example 8

I think that this is more clearly represented by Figure 4.7.

```
IF ( (LAT .GE. -30.0) .AND. (LAT .LE. 30.0) ) THEN
   IF (ST .LT. -30.0) THEN
      PRINT *,'Error: Equatorial temperature is too low, ST= ',ST
   ENDIF
ENDIF
```

Figure 4.7: Example 9

You could just as easily switch the order of the two IF statements, see Figure 4.8.

Keep in mind that when the first 'IF statement' is evaluated that if it is false then the rest of the block is ignored. So, your first 'IF statement' should be the most restrictive in this case. If you expect fewer regions to have a lower than -30 Celsius temperature than the number of regions with latitudes between -30 and 30 degrees, then Example 10 is more efficient.

```
      IF (ST .LT. -30.0) THEN
         IF ( (LAT .GE. -30.0) .AND. (LAT .LE. 30.0) ) THEN
            PRINT *,'Error: Equatorial temperature is too low, ST= ',ST
         ENDIF
      ENDIF
```

Figure 4.8: Example 10

Even more nested

Let's consider a case with more complicated nesting than Example 10. The problem will be similar to Example 10, but now we will be looking to highlight (not find errors) the longitude, LONGIT, and latitude, LAT, values that give a surface temperatures, ST, between 20 and 25 degrees Celsius. We only want to find longitude values between 0 and 40 degrees or between 60 and 70 degrees and latitude values between 22 and 27 degrees.

```
      IF ((ST .GE. 20.0) .AND. (ST .LE. 25.0)) THEN
         IF ((LONGIT .GE. 0.0) .AND. (LONGIT .LE. 40.0)) THEN
            IF ((LAT .GE. 22.0) .AND. (LAT. LE. 27.0)) THEN
               PRINT *,'Search for values of ST between 20 and 25 for certain longitude and
     &        latitudes'
               PRINT *,'is satisfied for, ST= ',ST,' LAT= ', LAT,' LONGIT= ',LONGIT
            ENDIF
         ELSE IF ((LONGIT .GE. 60.0) .AND. (LONGIT .LE. 70.0)) THEN
            IF ((LAT .GE. 22.0) .AND. (LAT. LE. 27.0)) THEN
               PRINT *,'Search for values of ST between 20 and 25 for certain longitude and
     &        latitudes'
               PRINT *,'is satisfied for, ST= ',ST,' LAT= ', LAT,' LONGIT= ',LONGIT
            ENDIF
         ENDIF
      ENDIF
```

Figure 4.9: Example 11

Note the lines in Example 11 that carried over to the next line are continued with an ampersand in column 6.

4.2 Input and output

You will now learn more about input and output from your program. Our programs won't always read input entered by the person running the program. Moreover, it is often more desirable to output results from a program directly into its own file, rather than to the screen for viewing. You may find a circumstance in which you aren't even logged into the computer but your program is running. In this case, you will

(a) want to have the program automatically read in values from a file, and

(b) output the results to a file so that you can view it once you have logged in again.

In this program, I have included the variable LAT just to let you see how two variable are read in and output (the LAT isn't used in any calculations).

```
      PROGRAM TMPCON
c     Written by Dr Steven Dobbie, Oct 1, 2002.
c     Computes the temp conversion from Celsius to Fahrenheit.
c     Version 3
      IMPLICIT NONE
      REAL TC,TF,LAT
      OPEN(UNIT=22,FILE='input.dat',STATUS='OLD')
      OPEN(UNIT=23,FILE='output.dat',STATUS='new',ACCESS='APPEND')
c     Assign a value for Temperature in Celsius
      PRINT *,'Please enter a temp in Celsius'
      READ (22,*) TC,LAT
c     Compute the value of Temp in Fahrenheit
      TF=9.0*TC/5.0+32.0
c     Output the results
      WRITE(23,*) 'Program: Temperature Converter for a specified latitude'
      WRITE(23,*) 'Celsius T= ',TC,' Fahrenheit T= ',TF,' Latitude LAT=',LAT
      CLOSE(22)
      CLOSE(23)
      STOP
      END
```

Figure 4.10: Example 12

4.2.1 OPEN statements

The OPEN statement links files to the program. The files may or may not exist at the time the program runs. If the program is reading input from a file then the open statement must appear before the read statement. In Example 12, line 7 opens a file called 'input.dat' and associates the number 22 with the file. The status equaling 'OLD' indicates that the file input.dat should be available in the directory that you run the program at the time of running. Line 11 is a read statement that has the number 22 present and so it will read from the file that is associated with the unit number 22, i.e. input.dat. The read statement will read in two values from the 'input.dat' file and the values will be assigned to the variable TC and LAT. The second OPEN statement is on line 8 and is associated with the unit number 23.

Unit number choice

You are free to choose the values of the unit number (like 22 and 23 in the last example) from values between 10 to 99. Some values are restricted, like 5, 6, etc. You should never open two files with the same unit number.

The write statements in Example 12 are on lines 15 and 16. Instead of the output being printed to the screen on these lines, it is printed into the file associated with the unit number 23, which is `output.dat`. If your program runs successfully, then list, *ls*, the contents of your directory and you will see a new file named `output.dat`. The `ACCESS='append'` tells the computer to add the output to the file at the end of the file. You see, the file may already exist when the program runs. If it does and has results already in it then this option will tell the computer to append the new results. If there isn't a file named `output.dat` in the directory, then it will create a new file and output the results to it.

Closing unit numbers

When you are finished inputting or outputting results, then you should always `CLOSE` the `OPEN` statement. The close statements for the unit numbers 22 and 23 are shown just before the end of the program in Example 12. Closing is, in general, issued by the command

```
CLOSE(unit number)
```

4.2.2 `OPEN` statement options

`STATUS` option

The `OPEN` statement has the option for `STATUS`. You will typically set this to be either `STATUS='OLD'`, `'NEW'` or `'UNKNOWN'`. If you use `'OLD'`, then the file should already exist in your directory in which you run the program. You can read in data from a file already in existence or you can output to this file depending on whether you open this file for input or output. If the file already exists and has data in it and you have linked it to your program as an output file, then you may or may not want to overwrite the data currently in the file. If you want to append new data to the file, then you should include `STATUS='OLD'`, `ACCESS='APPEND'` in the `OPEN` statement. If you don't care about the data currently in the output file then don't include the `ACCESS='APPEND'` statement and it will clear the contents in the output file when you write to the file.

If you use `STATUS='NEW'` in your `OPEN` statement, then there shouldn't be a file in your directory with the name used in the `OPEN` statement. This choice of `STATUS` is used only for output files, since you would expect to read input from a file that isn't in your directory. The `STATUS='UNKNOWN'` option is used when you don't know if there is a file with that name in your directory. This option can be used for `OPEN` statements linking to output files. You may or may not already have the output file created when you run the program. You should use `ACCESS='APPEND'` if you don't want to delete the results previously stored in that file. When using `STATUS='UNKNOWN'` for input files, you should be very careful about what you are doing. If you are unsure if the file will be there, then you are unsure if the input is available. If you use this option, then I recommend that you take advantage of error checking to whether or not the input is coming from a file (first) or from the terminal (as a backup).

Sequential and direct file access

When the computer accesses a file to read it, the default access is sequential (you can specify it as `ACCESS='SEQUENTIAL'` in the `OPEN` statement if you like). If you want to read line 154 (record 154), then with a sequential `OPEN` statement the computer will read through all of the records leading up to and including the 154th line, or record. If you open a file as `ACCESS='DIRECT'` (you have to set the `RECL` as well), then you can jump directly to record 154. The computer will calculate how many records to jump of length `RECL` and will then go directly. This is much faster if you have huge data sets that you only want to select some values from. The record length will vary from computer to computer, see Daniel Gordon's reference notes if you are interested in further reading.

`OPEN` errors

In your `OPEN` statement, you have the option of setting `ERR` and `IOSTAT` options. These are very useful and should be used. An example is as follows in Figure 4.11.

```
      REAL A,B,C
      INTEGER IERR
      OPEN(UNIT=35, FILE='input.dat', STATUS='OLD',
     & ERR=9035,IOSTAT=IERR)
      READ(35,*) A,B,C
      D=(A+B+C)/3.0
      STOP
C     Error trapping and printing section
9035  PRINT *,'ERROR: Input file error.  Error code=',IOSTAT
      STOP
      END
```

Figure 4.11: Example 13

The `OPEN` statement in Example 13 states that `STATUS='OLD'`, so the file called `input.dat` should exist in your directory when you run your program. If the input file isn't present in the directory, then the `OPEN` statement will have an error and will jump to line label 9035 (the error printing section). You can use `IOSTAT` to print out more detailed error statements. Please refer to Daniel Gordon's notes for details about the values of `IOSTAT`.

PARAMETER statement

Sometimes, physical constants change. For example, the solar constant isn't really a constant! If you write a program to determine how much radiation reaches the Earth's surface from the Sun, then the result will depend on the solar constant, how much radiation is coming from the Sun. Some radiation programs are 10 000 lines long and you don't want to search through the program to find out exactly what the solar constant currently is and change it. You may not find all occurrences for example and it makes the code less clear to read. You can define constants as parameters by the following statement

```
REAL SOLCON
PARAMETER(SOLCON=1642.0)
```

Now, anywhere that you would use the value of the constant you use `SOLCON` instead.

By specifying the `SOLCON` as a parameter, it tells the computer that the code shouldn't treat `SOLCON` as a variable, i.e. it cannot be assigned a new value in the program, it cannot be assigned values that are read in, etc. It is a fixed value throughout the program.

You can assign more constants in the same `PARAMETER` statement. For example,

```
REAL SOLCON, ALBCON
INTEGER NUMRAY
PARAMETER(SOLCON=1642.0, ALBCON=10.1, NUMRAY=200)
```

5 Arrays, Loops and GOTO Statements

Arrays

In a recent assignment, you were asked to write a program that computed the average value of six ozone concentrations. The programs that you wrote involved having to read in the six values that were entered manually by the program user. Most of you wrote the program in the following way to perform this task:

```
REAL OZONE1,OZONE2,OZONE3,OZONE4,OZONE5,OZONE6,OZAVG
READ *,OZONE1,OZONE2,OZONE3,OZONE4,OZONE5,OZONE6
```

This can be simplified by the use of arrays. An array is a collection of variables that will be used in a similar or coupled way. The following is the way you would write the equivalent code with the use of arrays.

```
REAL OZONE(6),OZAVG
READ *, OZONE(1),OZONE(2),OZONE(3),OZONE(4),OZONE(5),OZONE(6)
```

Arrays are *declared* with a label part, `OZONE`, and a number enclosed in brackets, `(6)`. The number is always an integer and it refers to the number of variables that the array will have, this array has six variables. After the declaration of the variable, you can use any of the variables by just referring to label and the number (in brackets) which is called the index, in this case from 1 to 6. So, above, the READ statement reads in six values and assigns the values to the variables, OZONE(1), OZONE(2), ..., OZONE(6).

It may seem like our only saving was in the shorter declaration statement. You will soon see that when arrays are combined with loops that many arduous operations and tasks can be reduced to a few lines of programming.

Arrays can be defined with not just one index but many. Say we have six ozone values corresponding to measurements at each of 1 km, 5 km, and 10 km above the ground. You would have a total of 18 ozone values, so you could define REAL OZONE(18), but it would be clearer to define a two-dimensional array as

```
      REAL OZONE(3,6)
```

Notice that the array has $3 \times 6 = 18$ variables under one name. There are two indices so you refer to the values of each variable as follows. You may want to set the initial values to some measure amounts that you have.

```
      REAL OZONE(3,6)
      OZONE(1,1)=40.0
      OZONE(1,2)=34.0
      OZONE(1,3)=24.0
      OZONE(1,4)=44.0
      OZONE(1,5)=43.0
      OZONE(1,6)=39.0
```

These would be the six values for one altitude, say at 1 km above the ground. You have to decide and keep in mind that the index 1 refers to the measurements at 1 km, an index of 2 refers to measurements at 5km, and an index of 3 refers to 10 km. It is helpful to provide a comment in the program explaining the arrangement. Continuing, the values of the concentrations at 5 km and 10 km would be as follows:

```
      OZONE(2,1)=10.0
      OZONE(2,2)=12.0
      ...
      ...
      OZONE(3,4)=200.0
      OZONE(3,5)=213.0
      OZONE(3,6)=217.0
```

If your measurements were recorded in a computer, then you can read them in automatically with loops as follows:

```
      DO 100 I=1,3    ! Levels 1=1km, 2=5km, 3=10km
         DO 200 J=1,6     ! Six measurements at each level
            READ(22,*) OZONE(I,J)
 200     CONTINUE
 100  CONTINUE
```

Similarly, you can have loops with more indexes. You could have these six measurements of ozone measurements at three levels, but now also at four different days. The ozone amount would be declared as

```
      REAL OZONE(3,6,4)
```

If the second measurement taken at the 10 km level on the fourth day has a value of 302, then you would set

```
      OZONE(3,2,4)=302.0
```

5.1 Loops

Loops are designed to allow you to repeat operations. If a lot of your calculations look the same, but are using different variables, then loops can be used usually with great ease. We will start with a very basic example. We want to print out the numbers 1 to 1000. If you needed a list of numbers such as this it would take you a considerable time, and then you would probably still not do it error free. Fortran can perform this task with ease. Consider the following piece of program:

```
      INTEGER I
      DO 100 I=1,1000
         PRINT *,I
 100  CONTINUE
```

This piece of the code tells the computer to start a `DO` loop on the second line, label it with a value `100`, and have an index, `I`, go from 1 to 1000. This program will loop around 1000 times between the `DO` statement and the `100 CONTINUE` statement, doing everything inside these two as it goes.

Let us look at it in detail. When the program reaches line 2, it starts a `DO` loop with a label `100`. On this line, the variable `I` is set to 1 and the program checks if `I` is equal to 1000. The program then proceeds to the next line. On the next line, the program will print the current value of `I`, which is 1. Upon doing that, it will go to the next line, which is the `100 CONTINUE` line. This line tells the program to go back to the second line because it has the corresponding label of `100`. The program will go back up to the second line of the program, increment `I` by one, to a value of 2, and will proceed to the next line as long as `I` isn't greater than 1000. If `I` is greater than 1000, then the program will jump back to the `100 CONTINUE` line and continue in the program after that (it doesn't go to line 3). Note that when `I` is greater than 1000 and the program is going to jump back to the `100 CONTINUE` statement, that this is after it has incremented `I` by one again. So after the `DO` loop is executed, and you program continues, the value of `I` will be 1001.

It is also important to note that loops can add things up, bit by bit on each pass of the loop. In this way, you can approximate integrals.

Let us return to the ozone example. Recall that it is as follows with no loops or arrays:

```
      REAL OZONE1,OZONE2,OZONE3,OZONE4,OZONE5,OZONE6,OZAVG
      READ *,OZONE1,OZONE2,OZONE3,OZONE4,OZONE5,OZONE6
```

We already showed that with arrays, this simplifies to

```
      REAL OZONE(6),OZAVG
      READ *, OZONE(1),OZONE(2),OZONE(3),OZONE(4),OZONE(5),OZONE(6)
```

Now if we include loops, then this can be simplified to

```
      REAL OZONE(6),OZAVG
      DO 101 I=1,6
        READ *, OZONE(I)
 101  CONTINUE
```

The main difference is that if you had a thousand variables to input, then it would be a very lengthy task to enter the values without the use of arrays or loops. With these techniques, it would take almost no additional programming space (increase 6 to a thousand).

Nested loops

You can nest loops within loops. Say you have a checkers board with red and black pieces on it. Pretend that you are in the middle of the game and so red and black pieces are scattered over the board and some have been removed from the game. If you assign a value of 1 to each red piece and 2 to each black piece, then you could represent the state of the game by an array

```
      INTEGER GAME(8,8)
```

You would have to specify at this point if red or black pieces are at certain locations. For example,

```
      GAME(1,1)=1
      GAME(1,2)=0
      GAME(1,3)=1
      ...
      ...
      GAME(4,5)=2
      GAME(4,6)=1
      ...
      ...
      GAME(8,5)=0
      GAME(8,6)=2
      GAME(8,7)=0
      GAME(8,8)=0
```

¿From this, you can see that red pieces are at (1,1), (1,3), (4,6), etc. and that black pieces are at (4,5), (8,6), etc. You could sum up all of the red and black pieces on the board using a two dimensional array as follows:

```
      INTEGER SUMRED,SUMBLA,GAME(8,8)
      SUMRED=0
      SUMBLA=0
      DO 100 I=1,8
         DO 200 J=1,8
            IF (GAME(I,J) .EQ. 1) THEN
               SUMRED=SUMRED+1
            ELSE IF(GAME(I,J) .EQ. 2) THEN
               SUMBLA=SUMBLA+1
            ELSE IF(GAME(I,J) .NE. 0) THEN
               PRINT *,'Error in the game array'
            ENDIF
 200  CONTINUE
 100  CONTINUE
      PRINT *,'Number of red pieces is ',SUMRED
      PRINT *,'Number of black pieces is ',SUMBLA
      STOP
      END
```

As you can see in the above example, it is fine to nest loops. The innermost loop will be executed first. For example, the `DO 100 I=1,8` loop will set `I` to 1 then the `DO 200 J=1,8` loop will set `J` to 1. The statements (`IF` block in this case) inbetween will be evaluated, then the `200 CONTINUE` statement will be reached. The computer will go back to the `DO 200 J=1,8` line and increment `J` to 2 and test to see if it is less than 8 (which it is) and continue in the `IF` block again. Looping around the `DO 200 J=1,8` to `200 CONTINUE` statements will occur until `J` is incremented to 9 and so the `J` is greater than 8 and the program jumps back to `200 CONTINUE` and proceeds to the next line, `100 CONTINUE`. The `100 CONTINUE` will return back up to the `DO 100 I=1,8` line, increment `I` to a value of 2 and test to see if `I` is greater than 8. If it isn't then the program will proceed to the next line. This next line is the `DO 200 J=1,8` line. Since we are approaching the line from above (and not from the `200 CONTINUE` statement), the program sets up this loop to start again, sets `J` to 1, compares `J` to 8, then proceeds to the `IF` block again.

You should understand the pattern now. For each time the `DO 100 I=1,8` loop increases `I` by 1 the `DO 200 J=1,8` will loop eight times. At the end, the following will happen. When the loop is for `I` equals 8 and `J` just reaches 8, the `200 CONTINUE` will jump the program back up to the `DO 200 J=1,8` line and increment `J` to 9. But since `J` is now greater than 8, the program returns to the `200 CONTINUE` line and progresses to the next line, which is `100 CONTINUE`. It is the last loop of the `I` index too, so when the program jumps back to the `DO 100 I=1,8` and increments `I` to 9, then the program will jump back to the line after the `100 CONTINUE` statement. At this point, we are have finished the nested `DO` loops. It is important to note that `I` and `J` both equal 9 from now on unless they are reassigned a new value.

Initialisation

If you declare a real variable as

```
      REAL AVAR
```

If you never set `AVAR` to be a number like `AVAR=10.0` or equal to another variable that has been assigned a number, then you may be surprised to find out that the number may have a value even if you didn't set it to anything! So, if you have an array of concentrations or ozone values, and you only assign values to some of them (perhaps you are lacking observed values for some), then you should be careful when you print out the full array. The array variables that were never assigned values will miraculously have values. The values might be huge or very small. So, when you declare variables, it is a good idea to set them to a value that will mean something to you. For instance, you may set an array of concentrations to zero to start with as

```
      INTEGER I,J,K
      REAL CONC(3,3,3)
      DO 1001 I=1,3
         DO 1002 J=1,3
            DO 1003 K=1,3
               CONC(I,J,K)=0.0
 1003 CONTINUE
 1002 CONTINUE
 1001 CONTINUE
```

This will initialise all of the 27 values of the `CONC` array to zero to begin with.

Modelling

You may think why would I tackle a problem with 1000 variables. Surely that would be such a complex task even with arrays and loops! Well, you may well find that you don't need a thousand different variables, but you probably will need a collection of variables with thousands of different values! What do we mean when we say a computer will model a situation, like the dispersal of pollutants in a river? We mean that we pretend that a portion of the river is subdivided into little cubes of space, small enough that the variation of the pollutant from one block to the next is not too large. We can label the blocks 1 to 1000 or we could say 10 blocks spanning East to West, 10 blocks spanning North to South, and 10 blocks from the bottom to the top of the river. This would total 1000 blocks that we have divided the river up into. You could then use an array to describe the concentration of pollutant at any point as follows. We would first define the concentration variable (3D array) as

```
      REAL CONC(10,10,10)
```

Then we could set box `1,1,1` to have a pollutant dumped in it, so `CONC(1,1,1)=500` ppm of a pollutant. All the other boxes would be zero, i.e.

```
      DO 100 I =1,10
         DO 200 J=1,10
            DO 300 K=1,10
               IF (I .EQ. 1  .AND.  J .EQ. 1  .AND. K .EQ. 1) THEN
                  CONC(1,1,1)=500.0
               ELSE
                  CONC(I,J,K)=0.0
               ENDIF
 300        CONTINUE
 200     CONTINUE
 100  CONTINUE
```

You see, in the above code, we have an initial modelled state of the river. To model the river in the future time and watch the concentrations vary, then you would have to add the influences of diffusion and dynamical transport. We will consider some cases later in this course.

5.2 GO TO statements

The GO TO statement is to be used very sparingly. It should only be used when it is absolutely necessary or enables significant processing speed enhancement but little change in the readability of the program. You can use a GO TO at any point in your program, but you should only use it for certain situations. You can use the GO TO statement when you have found that your data has unphysical values and you wish to jump out of that part of the program and go to an error printing section. This can be accomplished by other means though. You can also use a GO TO statement when you are in a DO loop and you are ready to terminate it (whereas continuing the DO loop would waste significant computer processing time). We will consider these two cases.

Consider that you have read in a value of A and it is supposed to be positive. The program in Example 14 uses a GO TO to go to the error printing section.

```
      REAL A
      READ *, A
      IF (A .LT. 0.0) THEN
         GO TO 9033
      ENDIF
      A=A*2.0
      ...
      STOP
 9033 PRINT *,'ERROR: input value of A is less than zero'
      STOP
      END
```

Figure 5.1: Example 14

The computer will only get to the error section if the GO TO statement is executed. The program will continue on as normal after line 9033 and won't return to the code between the GO TO statement and the line 9033 (in this case).

IF, but no THEN

There is a shorter version of the IF statement if you are not going to include ELSE IF or ELSE statements. This version is commonly used with error printing statements and GO TO statements. For example, in Example 14 you could write the IF statement as just

```
      IF (A .LT. 0.0) GO TO 9033
```

There is no need for an ENDIF if you don't include a THEN after the IF. Now consider the second example of a GO TO statement. This is used for early termination of a DO loop.

```
      REAL A,B
      INTEGER I, ITOT
      PARAMETER(B=10.0, ITOT=10000)
      A=0.
      DO 100 I=1,ITOT
         A=A+1.5
         IF (A .GT. B) THEN
            GO TO 101
         ELSE IF (I .EQ. ITOT) THEN
            PRINT *, 'Warning: Do Loop Reached upper limit'
         ENDIF
100   END DO
101   CONTINUE
      STOP
      END
```

Figure 5.2: Example 15

CONTINUE On?

In Example 15, variable A will exceed 10.0 (parameter B) very quickly and so to continue in the DO loop would waste a lot of time. The program breaks out of (terminates) the DO loop early by jumping to the CONTINUE statement on the line after the END DO statement. The CONTINUE statement in this case tells the program to continue to the next line of the program. The value of I will be the value it had when the loop was terminated. You should never use GO TO statements to jump lots of lines after the END DO. This will make the program difficult to read by you in the future or any person using your code.

6 Functions

Functions are either intrinsic or external. Intrinsic functions are available for your use without you having to write the program to perform the task; whereas, external functions are written by you.

6.1 Intrinsic functions

The following is a list of commonly used intrinsic functions.

Function	Comment
`SQRT(X)`	Square root of X
`ABS(X)`	Absolute value of X
`SIN(X)`	Sine of angle X (in radians)
`COS(X)`	Cosine of angle X (in radians)
`TAN(X)`	Tangent of angle X (in radians)
`EXP(X)`	e raised to the power of X (e = 2.71828...)
`LOG(X)`	Natural log of X
`LOG10(X)`	Log to the base 10 of X
`INT(X)`	Converts real variables to integer
`REAL(I)`	Converts integer to real variables
`MOD(I,J)`	Integer remainder of I/J

The following is a program that uses intrinsic functions `COS` and `SIN`:

```
REAL VAL,PI
INTRINSIC COS,SIN
PI=3.141592654          ! A value for pi
VAL= COS(PI/2.0)
PRINT *, VAL
PRINT *, COS(PI/2.0)
PRINT *, COS(PI)
PRINT *, SIN(PI)
PRINT *, SIN(PI/2.0)
STOP
END
```

Note: The '`!`' indicates that the remainder of that line is a comment.

The functions are declared as `INTRINSIC` after the declarations in the program. The functions are called, for example on the fourth line, with an argument passed to it which is the angle in radians. The fourth line states '`VAL=COS(PI/2.0)`' and so the `COS` function will be calculated using `PI/2.0` as the angle. The resulting `COS` of this angle is assigned to the variable `VAL`. Notice that you can use functions in print statements.

6.2 External functions

External functions are similar in the way they are implemented, but you have to write the subprogram to do the calculation. Consider the example below:

```
PROGRAM MAIN
IMPLICIT NONE
REAL X,Y,Z,HEIGHT
EXTERNAL HEIGHT
X=10.
Y=20.
Z=HEIGHT(X,Y)
PRINT *, X,Y,Z
STOP
END

REAL FUNCTION HEIGHT(X,Y)
REAL X,Y
HEIGHT=X**2+Y**2
RETURN
END
```

The main program is the first 10 lines starting with `PROGRAM MAIN`. This program takes values of `X` and `Y` and computes a `Z` value according to a formula. The formula is the addition of the square of `X` and the square of `Y`. You can see where it assigns `Z` to the function `HEIGHT` in the main program, `Z=HEIGHT(X,Y)`. On the third line, `HEIGHT` is declared to be a real number (you don't have to do this for intrinsic functions). On the fourth line, the function is stated to be external since this function is not something that is calculated as much as `COS` or `SIN` and so it isn't an `INTRINSIC` function i.e. part of the main fortran library of software. The function has two arguments, `X` and `Y`.

The function itself is five lines long and is outside of the main program. Above, it starts with declaration type (`REAL`), then `FUNCTION`, then the function name, `HEIGHT`, followed by the arguments that are passed to the function, `(X,Y)`.

You have to declare the arguments in the function as well as in the main program. The names need not be the same. In the function, you must set the function name, `HEIGHT`, equal to a value. In this function, the value is set equal to the square of `X` plus the square of `Y`. Functions are ended by `RETURN` and `END`. After the function has been calculated, the control will revert back to the program and assign the `VAL` variable to the value of the function.

Functions should only be used to calculate simple, single values.

6.3 Subroutines

Subroutines are extensively used in programs to enable a program structure to be broken into smaller logical pieces. Subroutines can be used to perform input tasks, do extensive calculations with multiple variables or arrays, print output, analyse trapped errors, etc.

Subroutines have the following form. In the program you will have

```
CALL NAME(arguments)
```

`CALL` tells fortran to go to the subroutine named `NAME` (you decide on the name), and it passes the arguments to the subroutine for use there. Like the function, the subroutine is stated to be `EXTERNAL` and it is placed outside of the main program. The subroutine takes the form

```
      SUBROUTINE NAME(arguments)
```

and ends with RETURN followed by END. It is best illustrated by an example. The following code is a simple example of a program that reads three numbers that you enter, it sorts them into ascending order, and then prints the results. The program currently loops five times.

```
      PROGRAM SWAPIT
      IMPLICIT NONE
      INTEGER A,B,C,I,NTOT
      EXTERNAL INP,SWAP
      DO 100 I=1,5
C        Input the numbers
         CALL INP(A,B,C)
C        Sort the numbers
         IF (A .GT. B) CALL SWAP(A,B)
         IF (A .GT. C) CALL SWAP(A,C)
         IF (B .GT. C) CALL SWAP(B,C)
         PRINT *,' '
         PRINT *,'RESULTS:'
         PRINT *,'In order, ',A,B,C
         PRINT *,' '
 100  CONTINUE
      STOP
      END

      SUBROUTINE INP(A,B,C)
      IMPLICIT NONE
      INTEGER A,B,C
      PRINT *,'Enter 3 positive integers (zeros will end)'
      READ *, A,B,C
      RETURN
      END

      SUBROUTINE SWAP(X,Y)
      IMPLICIT NONE
      INTEGER X,Y,TMP
      TMP=X
      X=Y
      Y=TMP
      RETURN
      END
```

There is one call in the main program to the subroutine INP. The INP subroutine is written to accept INPut values from the keyboard by the user. These values are then used by the main program. Lines 9 to 11 call the SWAP subroutine to switch the order of the values of A, B, and C depending on their magnitude. The results are printed after this swapping takes place. Note that IMPLICIT NONE should appear in every main program, subroutine, and function.

This program can be better written as follows:

```
      PROGRAM SORTIT
      IMPLICIT NONE
      INTEGER A,B,C,I,NTOT
      PARAMETER(NTOT=5)
      EXTERNAL INP,OUTRES,SWAP
      DO 100 I=1,NTOT
C        Input the numbers
         CALL INP(A,B,C)
C        Sort the numbers
         IF (A .GT. B) CALL SWAP(A,B)
         IF (A .GT. C) CALL SWAP(A,C)
         IF (B .GT. C) CALL SWAP(B,C)
C        Output the results
         CALL OUTRES(A,B,C)
 100  CONTINUE
      STOP
      END

      SUBROUTINE INP(A,B,C)
      IMPLICIT NONE
      INTEGER A,B,C,ITM,NNEG,ERR
      EXTERNAL TERM
      NNEG=0
 200  CONTINUE
      PRINT *,'Enter 3 positive integers (zeros will end)'
      READ (UNIT=5,FMT=*,ERR=300) A,B,C
      WRITE (UNIT=6,FMT=*,ERR=300) A,B,C
C     READ (5,FMT=*,ERR=300) A,B,C
      IF ((A .LT. 0) .OR. (B .LT. 0) .OR. (C .LT. 0)) THEN
         NNEG=NNEG+1
         IF (NNEG .EQ. 2) THEN
            ITM=5
            CALL TERM(ITM)
         ENDIF
         GO TO 200
      ENDIF
      IF ((A .EQ. 0) .AND. (B .EQ. 0) .AND. (C .EQ. 0)) THEN
         ITM=1
         CALL TERM(ITM)
      ENDIF
      NNEG=0
      RETURN
 300  CONTINUE
      ITM=3
      CALL TERM(ITM)
      END

      SUBROUTINE TERM(ITM)
      INTEGER ITM
      IF (ITM .EQ. 5) PRINT *,'TERMINATING: Too many neg inputs.
     &  Signal = ',ITM
      IF (ITM .EQ. 1) PRINT *,'TERMINATING NORMALLY FROM INPUTS SUB.
     &  Signal = ',ITM
      IF (ITM .EQ. 3) PRINT *,'TERMINATING: Inputs incorrectly entered
     &  Signal = ',ITM
      STOP
      RETURN
      END

      SUBROUTINE SWAP(X,Y)
      IMPLICIT NONE
      INTEGER X,Y,TMP
      TMP=X
      X=Y
      Y=TMP
      RETURN
      END

      SUBROUTINE OUTRES(A,B,C)
      IMPLICIT NONE
      INTEGER A,B,C
      PRINT *,' '
      PRINT *,'RESULTS:'
      PRINT *,'In order, ',A,B,C
      PRINT *,' '
      RETURN
      END
```

This program does much more checking and error trapping. I will highlight the main parts. The main program is at the top and is called `SORTIT`. I have defined a `PARAMETER NTOT` with a value of 5 instead of specifying the value in the program. This allows for easier changes. `INP` and `SWAP` are called in the main program in the same way as before, except now the printing is put in a subroutine called `OUTRES` (output results).

Looking at subroutine `INP`, you will notice that three integers are read in from the keyboard (`UNIT=5`) in any format (`FMT=*`), and if there is an error the program will jump to line label `300` (as indicated by the `ERR` value). This could happen if the person entering the values enters alphabetical characters or things that make no sense (e.g. `4-44`). If you don't want to trap errors, then `READ *, A, B, C` will read in the three values for you.

If you have an error inputting values, then the program will jump to line `300` and set `TIM=3` and call a subroutine called `TERM` (terminate). Terminate takes the `ITM` value (which I assigned differently for each error that occurs) and prints an output statement telling the program user what error occurred. Notice that `TERM` has a `STOP` in it and so the program will end there. A `STOP` placed anywhere will terminate the program running.

Back to the `INP` subroutine. Reading through, you should see that if the user enters three 0s then the program will set `ITM` to 1 and go to the subroutine `TERM` to terminate normally.

If the user enters negative numbers (we requested only positive) two times, then the program will know that because `NNEG` has been incremented by 1 each time the inputs were noted to be negative. The `TERM` will be called with a signal `ITM=5`. `TERM` prints out the message to the screen that the user keeps entering negative numbers and so has stopped the program running.

This last example shows that subroutines can call other subroutines (functions and subroutines cannot call themselves).

6.4 Compiling and linking

Subroutines and functions can be in different files. You can compile each file separately and then link them together as separate steps. Until now, you have been compiling and linking at the same time. Let us consider a simple example.

```
      PROGRAM SIMPLE
      INTEGER COUNT(5)
      REAL X,Y
      EXTERNAL CHGIT
      X=1.0
      Y=2.0
      COUNT(1)=1.0
      COUNT(2)=2.0
      COUNT(3)=3.0
      COUNT(4)=4.0
      COUNT(5)=5.0
      PRINT *, X,Y,COUNT(1),COUNT(2),COUNT(3),COUNT(4),COUNT(5)
      CALL CHGIT(X)
      PRINT *, X,Y,COUNT(1),COUNT(2),COUNT(3),COUNT(4),COUNT(5)
      END

      SUBROUTINE CHGIT(B)
      REAL B
      B=50.
      RETURN
      END
```

The main program is called `SIMPLE` and the subroutine is called `CHGIT`. The main program sets `X`, `Y`, and the five values of `COUNT` to certain values and prints them. The subroutine `CHGIT` is called which changes the value of `X` in the main program when it returns and then prints the values again. You could call the main program `part1.f` and the subroutine `part2.f`. You could compile (but not link) by typing

```
f77 -c part1.f
f77 -c part2.f
```

The `-c` indicates to compile but not link the file. These two compilations will produce

```
part1.o
part2.o
```

These can now be joined into one executable program by the following:

```
f77 part1.o part2.o  -lm
```

This will create an `a.out` file that is the same as if you had the main and subroutine in one file named `part1and2.f` and typed `f77 part1and2.f` to get `a.out`.

Why would I want to compile separately and then link?

In the future, you may work on a large code that is maintained by an organisation. I currently work on the UK Meteorological Office's boundary layer dynamics code which is over 50 000 lines long. If you compile and link in one command the whole code, then it can take up to a minute. This doesn't seem long, but if you are trying some new things then you may want to know as quickly as possible if the code will compile correctly. I usually only modify one subroutine and so compiling and linking the whole program is a waste of time. I have compiled all of the subroutines separately and now when I change things in that one subroutine then I just compile it and link it to the others. It is much quicker.

6.5 Common blocks

A subroutine that is called from the main program (or another subroutine) will often want to use variables that are contained in the main program (or calling subroutine). In the above, you see that these variables have to be included in the argument list. There is another way to make variables available, through common blocks. Consider the following example:

```
REAL A,B,C,X,V
EXTERNAL CAL
COMMON A,B,C
A=1.
B=2.
C=3.
X=4.
V=5.
CALL CAL(X,V)
PRINT *,A,B,C,X,V
STOP
END

SUBROUTINE CAL(X,V)
REAL A,B,C,X,V
COMMON A,B,C
X=A*B*C*V
RETURN
END
```

The main program calls subroutine CAL and passes the arguments X and V to the subroutine. The subroutine calculates a new value for X which is equal to A*B*C*V. How did the A, B and C become accessible in the subroutine? Because there is a COMMON statement after the declarations in both the main program (or calling subroutine) and the subroutine that is called. There can only be one COMMON in each subroutine.

You can also label common blocks. The form is

```
COMMON /label/ variables
```

Consider the following example:

```
PROGRAM COMEX
REAL A,B,C,X,QUADR,Q,S
COMMON A,B,C
COMMON /SOME/ Q
EXTERNAL QUADR
A=1.
B=2.
C=3.
X=4.
Q=5.
S=6.
PRINT *,A,B,C,X,Q,S
CALL INPUTS(X)
PRINT *,A,B,C,X,Q,S
Q=QUADR(X)
PRINT *,A,B,C,X,Q,S
CALL DISPLY(Q,X,S)
PRINT *,A,B,C,X,Q,S
STOP
END

SUBROUTINE INPUTS(X)
REAL A,B,C,X,Q
COMMON A,B,C
COMMON /SOME/ Q
PRINT *,'Enter values for a,b,c,x, and q please (all real)'
READ *, A,B,C,X,Q
RETURN
END

REAL FUNCTION QUADR(X)
REAL A,B,C,X
COMMON A,B,C
A=11.
B=12.
C=13.
X=14.
Q=15.
QUADR=20.
RETURN
END

SUBROUTINE DISPLY(Q,X,U)
REAL U,X
U=100.
RETURN
END
```

In this example, you can see that `COMMON A,B,C` appears in all of the subroutines that need those values. The main program also has a variable `Q` that is only needed in the subroutine `INPUTS`. `Q` could be added to the list of variables in `COMMON` but I have decided that since it is only used in the `INPUTS` subroutine that I will define a new `COMMON` block for it. It is called `COMMON /SOME/`. The main program and the subroutine `INPUTS` both have this and so the variable `Q` is accessible between both of those program components. Notice that common blocks don't have to appear in all subroutines, the last subroutine, `DISPLY`, only uses the passed arguments.

7 DO ENDDO, Characters and LOGICALs

7.1 DO ENDDO

In fortran, there is a progression away from the line labels. So in place of

```
      DO 100 I=1,100
         code...
 100  CONTINUE
```

there is an alternative. It is the `DO ENDDO` statement as shown in the next example.

```
      DO I=1,100
         code...
      ENDDO
```

Similarly for nested statements. The code

```
      DO 100 I=1, 100
         DO 200 J=1,20
            code...
 200     CONTINUE
 100  CONTINUE
```

is equally represented by

```
      DO I=1,100
         DO J=1,20
            code...
         ENDDO
      ENDDO
```

7.2 Characters

A new type of variable that we will introduce is the character variable. It is defined as `CHARACTER S` in the example below. To assign a value, you must put the value of the character in brackets, i.e. `S='A'`. The `PRINT *, S` will result in a character 'A' being printed out.

```
      IMPLICIT NONE
      REAL A
      CHARACTER S
      INTEGER I,J
      S='A'
      PRINT *, S
      STOP
      END
```

In the above example, only one character was held in the variable `S`. You can specify the length of the character variable by `CHARACTER*X`, where `X` is the length of the variable. If you want to hold 'ABCDE' in a character variable, then you need a length of five characters, as in the example below.

```
IMPLICIT NONE
REAL A
CHARACTER*5 S
INTEGER I,J
S='ABCDE'
PRINT *, S
S='AB DE'
PRINT *, S
STOP
END
```

In `OPEN` statements, instead of explicitly naming the input file, you can set it equal to a character variable (see example below). In this example, you enter the name of the file (that is not more than 20 characters in length) and it will assign the name to the `FILE` statement in the `OPEN` statement.

```
IMPLICIT NONE
REAL A
INTEGER I,J
CHARACTER*20 FNAME
PRINT *,'Please enter the filename'
READ *, FNAME
OPEN(UNIT=22,FILE=FNAME)
READ(22,*) A
PRINT *,A
STOP
END
```

Just like with real or integer arrays, you can define arrays of characters. In the example below, I have defined an array `FILEN` with five variables which each can hold 20 characters in their variable. The code below opens different input files and reads a number from each of them and prints them to the screen. Note that each file is closed before the next one is opened since it is using the same `UNIT`.

```
IMPLICIT NONE
REAL A
INTEGER I
CHARACTER*20 FILEN(5)
DO I=1,5
   IF (I .EQ. 1) FILEN(I)='file1.dat'
   IF (I .EQ. 2) FILEN(I)='file2.dat'
   IF (I .EQ. 3) FILEN(I)='file3.dat'
   IF (I .EQ. 4) FILEN(I)='file4.dat'
   IF (I .EQ. 5) FILEN(I)='file5.dat'
   OPEN(UNIT=22,FILE=FILEN(I))
   READ(22,*) A
   PRINT *,A
   CLOSE(22)
ENDDO
STOP
END
```

7.3 Logicals

I will now introduce another variable. This variable is called a logical, it can hold the value true or false only. The logical `ABLG` in the example below is declared as `LOGICAL ABLG`. Three lines down from that, it is set equal to `TRUE`. Be sure to use the periods on either side of the `TRUE` or `FALSE` when assigning a logical. The output from the print statements will be either `T` or `F`.

```
IMPLICIT NONE
REAL A,B
LOGICAL ABLG
A=10.0
B=0.0
ABLG=.TRUE.
PRINT *,'To begin with, ABLG is ',ABLG
IF (A .GT. B) THEN
   PRINT *,'Is A greater than B? That is ',ABLG
ENDIF
STOP
END
```

When using `IF` statements with logicals, then you need to use the `.EQV.` or `.NEQV.` for logicals. `.EQ.` and `.NEQ.` will not work with logicals.

```
IMPLICIT NONE
REAL ASIZ
INTEGER I
LOGICAL LARG
ASIZ=0.0
LARG=.FALSE.
IF (LARG .EQV. .FALSE.) PRINT *, 'The logical is False'
IF (LARG .NEQV. .TRUE.) PRINT *, 'The logical is False'
STOP
END
```

7.4 Implicit input and output

The input file '`inputfile.dat`' has 12 lines of input (12 numbers). You will notice that the read statement only loops over 3 values. The `READ` statement has an extra loop in `J` that appears with the `READ` statement `(A(I,J), J=1,4)`. The `I` will increment to 1 and the `J` will loop over the 4 values, then `I` will increment to 2 and `J` will loop over the four values, etc.

```
IMPLICIT NONE
REAL A(3,4)
INTEGER I,J
OPEN(UNIT=22,FILE='inputfile.dat')
DO I=1,3
   READ(22,*) (A(I,J), J=1,4)
ENDDO
DO I=1,3
   DO J=1,4
      PRINT *, A(I,J)
   ENDDO
```

```
ENDDO
STOP
END
```

We could read in all the 12 data values without a `DO` loop, as in the following example.

```
IMPLICIT NONE
REAL A(3,4)
INTEGER I,J
OPEN(UNIT=22,FILE='inputfile.dat')
READ(22,*) ((A(I,J), J=1,4), I=1,3)
DO I=1,3
   DO J=1,4
      PRINT *, A(I,J)
   ENDDO
ENDDO
STOP
END
```

You can also print out the variables using the same format, as shown below.

```
IMPLICIT NONE
REAL A(3,4)
INTEGER I,J
OPEN(UNIT=22,FILE='inputfile.dat')
READ(22,*) ((A(I,J), J=1,4), I=1,3)
PRINT *, ((A(I,J), J=1,4), I=1,3)
STOP
END
```

The extension to higher dimensions is straightforward. Below, I have shown the three-dimensional case.

```
IMPLICIT NONE
REAL A(3,4,5)
INTEGER I,J,K
OPEN(UNIT=22,FILE='inputfile5.dat')
READ(22,*) (((A(I,J,K), J=1,4), I=1,3), K=1,5)
PRINT *, (((A(I,J,K), J=1,4), I=1,3), K=1,5)
STOP
END
```

8 Unix Commands

8.1 Unix: Accessing your floppy disk

I am going to go through some basic commands and functions of unix that are useful for every day work on a computer. To access a floppy drive in unix, there are two methods.

Method 1

If you are on a linux computer, then look at the file named `/etc/fstab`. Consult your system administrator if you are not on a linux computer. There will be a line in this file that refers to `/dev/fd0`, as below

```
/dev/fd0        /mnt/floppy ext2    noauto      0 0
```

To link the floppy drive to the computer system for use, type

```
mount /mnt/floppy
```

Now you can list the floppy drive contents by typing

```
ls /mnt/floppy}
```

If you want to store a file on the floppy then type

```
cp file /mnt/floppy
```

If you want to copy a file named `filename.txt` from your floppy to your directory that you are currently in then type

```
cp /mnt/floppy/filename.txt  .
```

(don't forget the period at the end). After you are finished using the floppy drive then you need to un-mount the floppy disk before you extract it from the computer. Type

```
umount /mnt/floppy
```

to release the floppy drive from the system. You may see that the light on the floppy drive goes on and there is activity when you type the `umount` command. This is because linux leaves some copying to the floppy until it is necessary, and it becomes necessary when you type `umount`. Wait for the light to go off before you eject the disk.

Method 2

The second method is the use of `mtools` (this may or may not be available). Insert the floppy disk (MS dos formatted) and type `mdir a:` to see what is on the disk. You can see that this is emulating MS dos. To copy a file from your current directory to the floppy drive, type

```
mcopy file a:
```

(the floppy is referred to as `a:`). If you want to copy a file from the floppy to your current directory then type

```
mcopy a:filename.txt .
```

(need the period). To delete files, type

```
mdel a:filename.txt.
```

8.2 Redirection

Say you have some files in your current directory and you want to get a list of them. Type `ls` and it will list them, e.g. `file1 file2 file3`. If you want the list to be put in a file then you can do this by typing

```
ls  > outfile
```

This will print those file names in a file called `outfile`. The '`>`' is the redirection of the data (file names in this case) to the output file. Another example would be if you want the compilation and linking errors to go to a file called `out.err` then type

```
f77 file.f > out.err.
```

Now consider a program called `runit.f` that does the following:

```
      REAL A
      READ *,A
      PRINT *, A
      STOP
      END
```

This program expects data to be entered when the program is run. So compile and link it and run the program.

```
f77 runit.f
```

If you run `a.out` then the program will wait until you enter the value for `A`. When you enter a value, like `54.0`, then it proceeds to print it out to the screen and the program ends. Alternatively, you could put `54.0` in a file called `inputfile.txt` and have the value in the file automatically read in using redirection when the program is run. Try it,

```
a.out < inputfile.txt
```

The program will take the value to be read in from the `inputfile.txt` file and it will print the value out and terminate normally.

8.3 Pipes

Sometimes, data files are so large that they take substantial time to be viewed by *vi* or other editors. In some cases, the data files exceed the limit that is viewable. In some cases, you want to see a sample of the data to see if it looks in good form. You can do this by using the '*more*' utility. Type `more datafile.dat` and it will show you one page full of the data file (without entering the full data file into memory).

If when you type *ls* you have too many files to view so that they screen past, then you should use '*more*' to handle this. Type `ls | more`. The vertical bar indicates that the results from the activity on the left (it ls) are fed into the command on the right (*more*). By doing this, *more* accepts the data from *ls* and puts only one page full on the screen at a time.

8.4 Other useful unix commands

it grep and *find* are two very useful unix commands of which I will only hint at their full use here. Type

```
grep string filename
```

and *grep* will search a file called `filename` to see if it contains the string of characters that you put in '`string`'. For example, if you want to look in a file `prgm.f` for all the subroutines, then type

```
grep subroutine prgm.f
```

and it will list all the lines that contain the word `subroutine`. This is an easy way of seeing how many subroutines are used in a program. If you are looking for a string but you don't know what file it is in then type

```
grep string *
```

The * will tell *grep* to look at all files in the current directory.

To find a file somewhere in your current directory or directories down from your current directory, then type

```
find . -name string -print
```

So if you are looking for the file named `lostfile.dat` then type

```
find . -name lostfile.dat -print
```

and it will list the location of any file (or files) named `lostfile.dat`.

9 Strange Ways of Passing Arrays to a Subroutine

Consider the main program

```
REAL A(10,20,30)
CALL SUBCL(A)
STOP
END
```

and the subroutine

```
SUBROUTINE SUBCL(B)
REAL B(10,20)
RETURN
END
```

Notice that the array in the subroutine (`B`) has one less dimension than the array in the main program (`A`). What happens in this circumstance? Well, all of the values for the first two dimensions of `A` are passed to `B`, whilst the third dimension of `A` is fixed at 1. So

```
PRINT *, ((A(I,J,1), I=1,10), J=1,20)
```

and

```
PRINT *, ((B(I,J), I=1,10), J=1,20)
```

gives the same results. So it is important to give some thought as to the order of the indexes in arrays that you create. If your program can run in three dimensions but usually runs in two dimensions, then you should specify the last index in arrays to be the dimension that isn't used in two-dimensional runs. Then you can pass slices of the array to subroutines when running in three dimensions and when running in two dimensions it just eliminates the loop over the slices (only does one slice).

10 Calculating Derivatives in Programs

A very basic way to calculate a derivative $\frac{\mathrm{d}f}{\mathrm{d}x}$ is to take finite differences of $\mathrm{d}f$ and $\mathrm{d}x$. Consider an example where $f = \cos x$. The following program will construct an array called `F(I)` that contains the values of the function `cos(x)`:

```
X=0.0
DX=2.0*3.141592654/100.0
DO I=1,100
   F(I)=COS(X)
   X=X+DX
ENDDO
```

This makes the `X` range split into 100 slices. With this function now determined, the derivative can be approximated by the following code:

```
DO I=1,99
   DERIV(I)=(F(I+1)-F(I)) / DX
ENDDO
```

You should try smaller and smaller values of `DX` and test to make sure that the derivatives are largely unaffected by the change.

11 Equivalence

You can have two or more variables (both either real, integer, or characters) refer to the same memory location to get its value. This is done by equivalence. Consider the following:

```
INTEGER H,D
EQUIVALENCE (H,D)
H=42
D=31
PRINT *, H
```

The value that is printed for `H` is 31. This is because `D` and `H` refer to the same location in memory and so if you change `H` or `D` then it changes the other variable because they both refer to the same memory location. The above example was for integers, you can also equivalence real variables and characters (don't mix the variable types in an equivalence though). The following is an example of a real equivalence:

```
REAL M,F
EQUIVALENCE (M,F)
M=42.0
F=F+1.0
PRINT *,M
```

The value printed will be `43.0` since the value was changed by the `F=F+1.0` line. Note that `F` didn't need to be initialised to `42.0` since `M` did that for `F` already.

11.1 Equivalence cautions

- Two variables in different common blocks cannot be equivalenced.
- Characters cannot be equivalenced with non-characters. Real and integer can be equivalenced but be careful, the result can in cases be machine dependent.
- Cannot equivalence to two different places at the same time.

11.2 Arrays

You can also equivalence arrays. For example,

```
REAL X,A(100),AN(100)
EQUIVALENCE (X,A,AN)
```

The result is that the same memory location is used for `X`, `A(1)`, and `AN(1)` and that all values of `A` use the same memory locations as `AN`. If you do

```
REAL X,A(100),AN(100)
EQUIVALENCE (X,A(2),AN)
```

Then the same memory location is used for `X`, `A(2)`, and `AN(1)`, but no other memory locations overlap between `A` and `AN`.

12 Data Statements

You can initialise data in your program in the following way:

```
      DATA X,Y / 5.0, 6.0 /
```

When the program is compiled (and only when it is compiled) the value of `X` and `Y` are set to `5.0` and `6.0`, respectively. `DATA` statements can be placed anywhere in the program but must be after the declarations and common blocks that use the variables. You can initialise arrays as well using `DATA` statements:

```
      INTEGER I, COUNT(5)
      REAL X,Y
      DATA X,Y / 5.0, 6.0 / COUNT / 7,6,3,4 /
      PRINT *, X,Y                 ! values of 5.0 and 6.0
      PRINT *, (COUNT(I), I=1,4)      ! values of 7 6 3 4
```

Now consider the following code:

```
 10   CONTINUE
      DATA P /1/
      PRINT *, P
      P=P+1
      IF (P .LE. 3) GO TO 10
```

Is this loop infinite? No, remember that the `DATA` statement is only implemented at compilation time. The initial value of `P` is 1 but when the program runs the value is never reset by a `DATA` statement.

13 Block Data

You can put all of the data statements in a 'SUBROUTINE' of their own. It is automatically executed at compilation time. This is the only way to initialise data in common blocks.

```
BLOCK DATA
INTEGER A
REAL W
COMMON  A, W
DATA A / 0 /, W / 0.0 /
END
```

This will initialise the variable A and W both to zero when the program begins to run. If these values are modified during the run then they will stay modified. This 'SUBROUTINE' is not executed again. DATA statements are usually reserved for constants in calculations that are not expected to change.

14 Random Numbers

For linux, `RAND()` is a function that returns a real value between 0.0 and 1.0 (including the 0.0 and 1.0 too). Every time `RAND()` is called, it returns a different value between those limits. If you look at the values that `RAND()` generates, then they are apparently randomly distributed over the region 0.0 to 1.0. You can visualise that the computer has a very long list of numbers that look random. If you wish to call `RAND()` five times to get five numbers, then you may want a different sequence of numbers if you were to run a program a second time. To change the list of random numbers that you get, you need to specify an initial seed to tell the random number generator where to start at a different point in the list of numbers. For linux computers,

```
INTEGER ISEED
REAL A
ISEED=30948
A=RAND(ISEED)
DO I=1,4
   A=RAND()
   PRINT *, A
ENDDO
```

If you run this program then it will print four random numbers. Run it again and it will print the same numbers. However, if you change that number `ISEED` (which was chosen at will) to a different value and then run the program again, then the program will give you four different numbers. Note, the random number generator will be used in assignment 6 and so if you are not using a linux computer then please consult Dr Wen or me as to the name of the random number generator (i.e. equivalent of `RAND()` for other systems). Note that a couple of the examples in these notes (e.g. equivalence) were taken from D. D. McCracken and W. I. Salmon, (1998). *Computing for Engineers and Scientists with FORTRAN 77*, 2nd Edn. John Wiley and Sons, Toronto.