Interactive Fortran 77
A Hands on Approach

Second edition

Ian D Chivers

Jane Sleightholme

Information about the Fortran 90 version is available at

http://www.kcl.ac.uk/kis/support/cc/fortran/f90home.html

to Joan and Martin
to Mark and Jonathan
to Glasgow

‘Flourish’

Preface to First Edition

The aim of this book is to introduce the concepts and ideas involved in problem solving with Fortran 77 using an interactive timesharing computer system. The book tries to achieve this using the established practices of *structured* and *modular* programming. Two techniques of problem solving, so-called *top-down* and *bottom-up* are also introduced.

The book has been developed from a one week full-time course on programming, given several times a year at Imperial College to a variety of students, both undergraduate and postgraduate. The course itself is a mixture of

- Lectures
- Tutorials
- Terminal sessions
- Reading

All work on the course is done in small groups, and the students have the option of working in pairs. Initially, students are shy about showing their ignorance, but quickly overcome this and learn a lot by helping one another out and articulating their problems. This is regarded as an essential part of the course.

The student is assumed to complete a minimum number of the problems. Experience on courses over several years has shown the authors that only by completing problems fully does the student get a realistic idea of the process of problem solving using a programming language. It is therefore recommended that all problems attempted are completed. Certain of the problems are used as a basis for further development in the course. This helps to reinforce the ideas of problem solving introduced earlier.

The authors are pleased to provide more details of the course to interested parties.

Ian D. Chivers
Malcolm W. Clark

1984

Preface to Second Edition

As most teachers know their ideas of how to approach a subject gradually change with time, for a variety of reasons. This edition reflects changes in four main areas

- a complete rewrite of the problem solving chapter;
- a new chapter on programming languages with an extensive bibliography; firstly as background material for the inquisitive reader; secondly to show the way Fortran has evolved and is still evolving by the incorporation of modern language constructs. This is becoming increasingly necessary given the current state of the proposed Fortran 8x standard;
- an alternate introduction to arrays more appropriate to a wider range of students.
- a complete revamp of Appendix E, to provide a complete list of functions in Fortran 77 with descriptions and examples.

Minor changes have been made throughout the book, reflecting the feedback we have had from the students over the years, at a number of colleges.

There are of course several corrections, and we are thankful to the many students who have pointed them out with great relish! We expect the same enthusiasm from students in pointing out the mistakes in this edition.

The first edition was prepared and typeset using the Draft Format text processing software running on a variety of CDC Cyber 6000 Series computers at Imperial College. Final output was to an APS μ 5 typesetter.

The Draft Format version was then transfered to an IBM PS/2 Model 60 running Ventura Publisher. Original output was to a variety of postscript laser printers, and final camera ready copy was obtained using the Linotron 300 typesetter at the University of London Computer Centre.

Our thanks to the students at King’s College for their comments on the drafts of this edition, and to UNEP for the use of a variety of facilities at the Monitoring and Assessment Research Centre, London, whilst on a very stimulating and enjoyable secondment.

Ian D. Chivers
Jane M. Sleighthome

1990

Table of Contents

Table of Contents

1

Introduction to computing

'Don't Panic'

Douglas Adams, 'The Hitch-Hiker's Guide to the Galaxy'

Aims

The aims of this chapter are to introduce the following:–

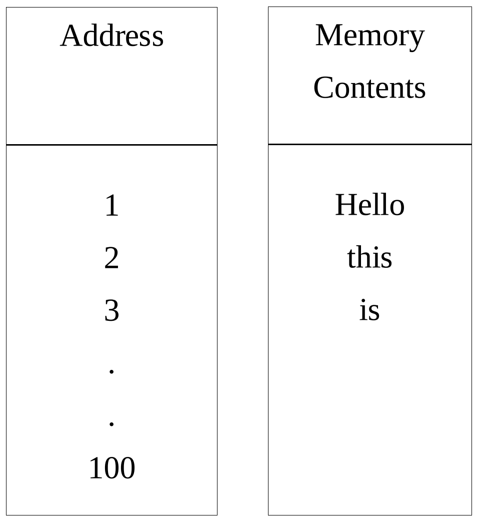
- the components of a computer — the hardware;
- the component parts of a complete computer system — the other devices that you need to do useful work with a computer;
- the software needed to make the hardware do what you want.

A computer

A computer is an electronic device, and can be thought of as a tool, like the lever or the wheel, which can be made to do useful work. At the fundamental level it works with *bits* (binary digits or sequences of zeros and ones). Bits are often put together in larger configurations, e.g. 8, 16, 32, 60, or 64. Hence computers are often referred to as 8-bit, 16-bit, or 32-bit, 60-bit or 64-bit machines.

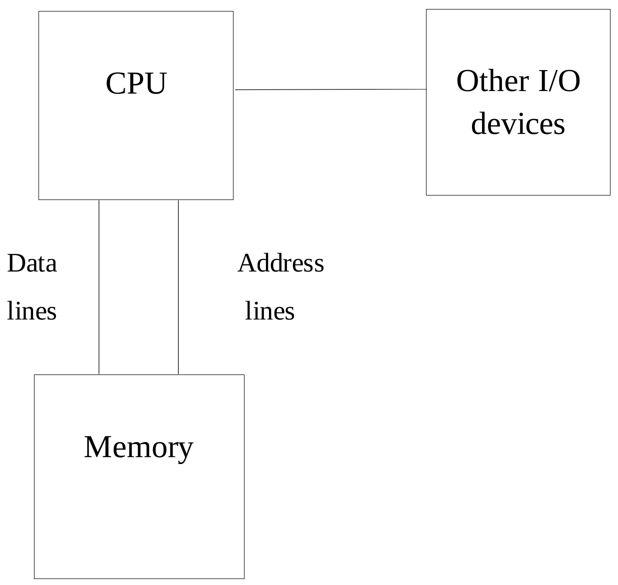
Most computers consist of the following:–

CPU	This is the brains of the computer. CPU stands for *central processor unit*. All of the work that the computer does is organised here.
MEMORY	The computer will also have a memory. Memory on a computer is a solid state device that comprises an ordered collection of bits/bytes/words that can be read or written by the CPU. A byte is generally 8 bits (as in *8-bit byte*), and a word is most commonly accepted as the minimum number of bits that can be referenced by the CPU. This referencing is called *addressing*. The memory typically contains programs and data. The following diagram illustrates the two ideas of address and contents of the memory at that address.



Word sizes of 8, 16 and 32 bits are commonly found in micro-computers; 16 and 32 bits are common for mini-computers; 32, 60 and 64 bits are common for mainframes. A computer memory is often called random access memory, or RAM. This simply means that the access time for any part of memory is the same; in order to examine location (say) 97, it is not necessary to first look through locations 1 to 96. It is possible to go directly to location 97. A slightly better term might have been *access at random*. The memory itself is highly ordered.

BUS A *bus* is a set of connections between the CPU and other components. The bus will be used for a variety of purposes. These include address signals which tell the memory which words are wanted next; data lines which are used to transfer data to and from memory, and to and from other parts of the computer system. This is typical of many systems, but systems do vary considerably; while the information above may



not be true in specific cases, it provides a general model.

A diagram for the constituent parts of a *typical* computer is given below.

The components of a computer system

So far the computer we have described is not sufficiently versatile. We have to add on other pieces of electronics to make it really useful.

Disks

	These are devices for storing collections of *bits*, which are inevitably organised in reality into bytes and files. One advantage of adding these to our computer system is that we can go away, switch the machine off, and come back at a later time and continue with what we were doing. Memory is expensive and fast whereas disks are slower but cheaper. Most computer systems balance speed against cost, and have a small memory in relation to disk capacity. Most people would be familiar with the two main type of disks on micro computers, and these are floppy disks, and hard disks. Micro floppy disks come in two main physical sizes, 5 1/4 and 3 1/2 inch. Hard disks are inside the system, and most people do not see this type of disk.
Tapes	These are slower than disks but cheaper, generally. They vary from ordinary, domestic cassettes used with micros to very large drives found on most mainframe systems. These devices are used for storing large quantities of data.
Others	There are a large number of other input and output devices. These vary considerably from system to system depending on the work being carried out. Most large computer systems have line-printers and laser printers whilst other installations may have more sophisticated i/o devices, e.g. plotters for the production of graphical output and photo-type-setters for the production of high quality printed material.

The most important i/o device is the terminal. This book has been written assuming that most of your work will be done at a terminal. Terminals tend to come in two main types — either a so called dumb terminal or a micro-computer with suitable terminal emulation (DEC VT100 is very common). In either case you communicate through the keyboard. This looks rather like an ordinary typewriter keyboard, although some of the keys are different. However, the location of the letters, numbers and common symbols is fairly standard. Don't panic if you have never met a keyboard before. You don't have to know much more than where the keys are. Few programmers, even professionals, advance beyond the stage of using two index fingers and a thumb for their typing. You will find that speed of typing is rarely important, it's accuracy that counts.

One thing that people unfamiliar with keyboards often fail to realise is that what you have typed in is not sent to the computer until you press the *carriage return* key. To achieve any sort of communication you must press that key; it will be somewhere on the right hand side of the keyboard, and will be marked *return, c/r, send, enter,* or something similar.

Software

So far we have not mentioned software. Software is the name given to the programs that *run* on the hardware. Programs are written in *languages*. Computer languages are frequently divided into two categories; *high-level* and *low-level*. A low level language (e.g. assembler) is closer to the hardware, while a high level language (e.g. Fortran) is closer to the problem statement. There is typically a one to one correspondence between an assembly language statement and the actual hardware instruction. With a high level language there is a one to many correspondence; one high level statement will generate many machine level instructions.

A certain amount of general purpose software will have been provided by the manufacturer. This software will typically include the basic operating system, one or more *compilers*, an *assembler*, an *editor*, and a *loader* or *link editor*.

- A *compiler* translates high level statements into machine instructions;
- An *assembler* translates low level or assembly language statements into machine instructions;
- An *editor* makes changes to another program;
- A *loader* or *link editor* takes the output from the compiler and completes the process of generating something that can be executed on the hardware.

These programs will vary considerably in size and complexity. Certain programs that make up the operating system will be quite simple and small (like copying utilities), while certain others will be relatively large and complex (like a compiler).

In this book we concentrate on software or programs that you write for your course, research, or work. As the book progresses you will be introduced to ways of building on what other people have written, and how to take advantage of the vast amount of software already written, tested and documented.

Operating systems

There are generally a variety of operating systems available for a particular computer. The choice of operating system will depend on the kind of work that the computer system has to do. A time- sharing operating system is one of the best for general purpose problem solving. These systems allow tens or even hundreds of users to use the system simultaneously. Rapid feedback is possible, and you can model complex systems, interact with the model, and even change the model, sometimes in a matter of minutes. It is also possible to set up a problem quickly and have it run as a background process, whilst you work on another aspect of the problem.

Problems

1. Distinguish between a memory address and memory contents.

2. What does RAM stand for?

3. What would a WOM (write only memory) do? How would you use it?

4. What does CPU stand for? What does it do?

2

Introduction to Problem Solving

They constructed ladders to reach to the top of the enemy's wall, and they did this by calculating the height of the wall from the number of layers of bricks at a point which was facing in their direction and had not been plastered. The layers were counted by a lot of people at the same time, and though some were likely to get the figure wrong the majority would get it right... Thus, guessing what the thickness of a single brick was, they calculated how long their ladder would have to be.

Thucydides, The Peloponenesian War

'When I use a word,' Humpty Dumpty said, in a rather scornful tone, 'it means just what I choose it to mean - neither more nor less'
'The question is,' said Alice, 'whether you can make words mean so many different things.'

Lewis Carrol, Through the Looking Glass and What Alice found there.

Aims

The aims are:–

- to examine some of the ideas and concepts involved in problem solving;
- to introduce the concept of an algorithm;
- to introduce two ways of approaching algorithmic problem solving;
- to introduce the ideas involved with systems analysis and design, i.e. to show the need for pencil and paper study before using a computer system.

Introduction

It is informative to consider some of the dictionary definitions of problem

- a matter difficult of settlement or solution, *Chambers*
- a question or puzzle propounded for solution, *Chambers*
- a source of perplexity, *Chambers*
- doubtful or difficult question, *Oxford*
- proposition in which something has to be done, *Oxford*
- a question raised for enquiry, consideration, or solution, *Webster's*
- an intricate unsettled question, *Webster's*

and a common thread seems to be a question that we would like answered or solved. So one of the first things to consider in problem solving is how to pose the problem. This is often not as easy as is seems. Two of the most common methods are

- in natural language
- in artificial language or stylised language

Both methods have their advantages and disadvantages.

Natural Language

Most people use natural language and are familiar with it, and the two most common forms are the written and spoken word. Consider the following language usage

- the difference between a three year old child and an adult describing the world;
- the difference between the way an engineer and a physicist would approach the design of a car engine;
- the difference between a manager and worker considering the implications of the introduction of new technology;

Great care must be taken when using natural language to define a problem and a solution. It is possible that people use the same language to mean completely different things, and one must be aware of this when using natural language whilst problem solving.

Natural language can also be ambiguous

Old men and women eat cheese.

Are both the men and women old?

Artificial Language

The two most common forms of artificial language are technical terminology and notations. Technical terminology generally includes both the use of new words and alternate use of existing words. Consider some of the concepts that are useful when examining the expansion of gases in both a theoretical and practical fashion

- temperature
- pressure
- mass
- isothermal expansion
- adiabatic expansion

Now look at the following

- a chef using a pressure cooker
- a garage mechanic working on a car engine
- a doctor monitoring blood pressure
- an engineer designing a gas turbine

Each has a particular problem to solve, and each will approach their problem in their own way; thus they will each use the same terminology in slightly different ways.

Notations

Some examples of notations are

- algebra
- calculus
- logic

All of the above have been used as notations for describing both problems and their solutions.

Resume

We therefore have two ways of describing problems and they both have a learning phase until we achieve sufficient understanding to use them effectively. Having arrived at a satisfactory problem statement we next have to consider how we get the solution. It is here that the power of the algorithmic approach becomes useful.

Algorithms

An algorithm is a sequence of steps that will solve part or all of a problem. One of the most easily understood examples of an algorithm is a recipe. Most people have done some cooking, if only making toast and boiling an egg.

A recipe is made up of two parts

- a check list of things you need
- the sequence or order of steps

Problems can occur at both stages, e.g. finding out halfway through the recipe that you do not have an ingredient or utensil; finding out that one stage will take an hour when the rest will be ready in ten minutes. Note that certain things can be done in any order – it may not make any difference if you prepare the potatoes before the carrots.

There are two ways of approaching problem solving when using a computer. They both involve *algorithms*, but are very different from one another. They are called *top-down* and *bottom-up.*

Top Down

In a *top down* approach the problem is first specified at a high or general level: prepare a meal. It is then refined until each step in the solution is explicit and in the correct sequence, e.g. peel and slice the onions, then brown in a frying pan before adding the beef. One drawback to this approach is that it is very difficult to teach to beginners because they rarely have any idea of what *primitive* tools they have at their disposal. Another drawback is that they often get the sequencing wrong, e.g. *now place in a moderately hot oven* is frustrating because you may have not lit the oven (sequencing problem) and secondly because you may have no idea how hot *moderately hot* really is. However as more and more problems are tackled top-down becomes one of the most effective methods for programming.

Bottom up

Bottom-up is the reverse to top-down! As before you start by defining the problem at a high level, e.g. prepare a meal. However, now there is an examination of what tools etc you have available to solve the problem. This method lends itself to teaching since a repertoire of tools can be built up and more complicated problems can be tackled. Thinking back to the recipe there is not much point trying to cook a six course meal if the only thing that you can do is boil an egg and open a tin of beans. The bottom-up approach thus has advantages for the beginner. However there may be a problem when no suitable tool is present. One of the authors' friend's learned how to make Bechamel sauce, and was so pleased by his success that every other meal had a course with a bechamel sauce. Try it on your eggs one morning. Here was a case of specifying a problem *prepare a meal*, and using an inappropriate but plausible tool *Bechamel Sauce.*

The effort involved in tackling a realistic problem, introducing the constructs as and when they are needed and solving it is considerable. This approach may not lead to a reasonably comprehensive coverage of the language, or be particularly useful from a teaching point of view. Case studies do have great value, but it helps if you know the elementary rules before you start on them. Imagine learning French by studying Balzac, before you even look at a French grammar. You can learn this way but even when you have finished, you may not be able to speak to a Frenchman and be understood. A good example of the case study approach is given in the book *Software Tools*, by Kernighan and Plauger.

In this book our aim is to gradually introduce more and more tools until you know enough to approach the problem using the top-down method, and also realise from time to time that it will be necessary to develop some new tools.

Stepwise Refinement

Both the above techniques can be combined with what is called *step-wise refinement*. The original ideas behind this technique are well expressed in a paper by Wirth entitled *Program Development by Stepwise Refinement,* published in 1971. This means that you start with a global problem statement and break the problem down in stages, into smaller and smaller sub-problems, that become more and more amenable to solution. When you first start programming the problems you can solve are quite simple, but as your experience grows you will find that you can handle more complex problems.

When you think of the way that you solve problems you will probably realise that, unless the problem is so simple that you can answer it straight away some thinking and pencil and paper work is required. An example that some may be familiar with is in practical work in a scientific discipline, where coming unprepared to the situation can be very frustrating and unrewarding. It is therefore appropriate to look at ways of doing analysis and design before using a computer.

Systems Analysis and Design

When one starts programming it is generally not apparent that one needs a methodology to follow to become successful as a programmer. This is generally because the problems are reasonably simple, and it is not necessary to make explicit all of the stages one has gone through in arriving at a solution. As the problems become more complex it is necessary to become more rigorous and thorough in ones approach, to keep control in the face of the increasing complexity and to avoid making mistakes. It is then that the benefit of systems analysis and design becomes obvious. Broadly we have the following stages in systems analysis and design

- Problem definition
- Feasibility study and fact finding
- Analysis
- Initial system design
- Detailed design
- Implementation
- Evaluation
- Maintenance

and each problem we address will entail slightly different time spent in each of these stages. Let us look at each stage in more detail.

Problem Definition

Here we are interested in defining what the problem really is. We should aim at providing some restriction on both the scope of the problem, and the objectives we set ourselves. We can use the methods mentioned earlier to help out. It is essential that the objectives are

- clearly defined;
- when more than one person is involved, understood by all people concerned, and agreed by all people concerned;
- realistic.

Feasibility Study and Fact Finding

Here we look to see if there is a feasible solution. We would try and estimate the cost of solving the problem and see if the investment was warranted by the benefits, i.e. cost benefit analysis.

Analysis

Here we look at what must be done to solve the problem. Note we are interested in finding what we need to do, but we do not actually do it at this stage.

Design

Once the analysis is complete we know what must be done, and we can proceed to the design. We may find there are several alternatives, and we thus examine alternate ways in which the problem can be solved. It is here that we use the techniques of top-down and bottom-up problem solving, combined with step-wise refinement to generate an algorithm to solve the problem. We are now moving from the logical to the physical side of the solution. This stage ends with a choice between one of several alternatives. Note that there is generally not one ideal solution, but several, each with their own advantages and disadvantages.

Detailed Design

Here we move from the general to the specific, The end result of this stage should be a sufficiently tightly defined specification to generate actual program code from.

It is at this stage that it is useful to generate *pseudo-code*. This means writing out in detail the actions we want carried out at each stage of our overall algorithm. We gradually expand each stage (step-wise refinement) until it becomes Fortran – or whatever language we want in fact.

Implementation

It is at this stage that we actually use a computer system to create the program(s) that will solve the problem. It is here that we actually need to know sufficient about a programming language to use it effectively to solve our problems. This is only one stage in the overall process, and mistakes at any of the stages can create severe difficulties.

Evaluation and testing

Here we try to see if the program(s) we have produced actually do what they are supposed to. We need to have data sets that enable us to say with confidence that the program really does work. This may not be an easy task, as quite often we only have numeric methods to solve the problem, which is why we are using the computer to solve the problem — hence we are relying on the computer to provide the proof; i.e. we have to use a computer to determine the veracity of the programs – and as Heller says *Catch 22.*

Maintenance

It is rare that a program is run once and thrown away. This means that there will be an on going task of maintaining the program, generally to make it work with different versions of the operating system, compiler, and to incorporate new features not included in the original design. It often seems odd when one starts programming that a program will need maintenance as we are reluctant to regard a program in the same way as a mechanical object like a car that will eventually fall apart through use. Thus maintenance means keeping the program working at some tolerable level, with often a high level of investment in manpower and resources. Research in this area has shown that anything up to 80% of the manpower investment in a program can be in maintenance.

Conclusions

A drawback, inherent in all approaches to programming, and to problem solving in general, is the assumption that a solution is indeed possible. There are problems which are simply insoluble – not only problems like balancing a national budget, weather forecasting for a year, or predicting which radioactive atom will decay, but also problems which are apparently computationally solvable. Knuth gives the example of a chess problem – determining whether the game is a forced victory for white. Although there is an algorithm to achieve this, it requires an inordinately large amount of time to complete. For practical purposes it is unsolvable. Other problems can be shown mathematically to be without solution. As far as possible we will restrict ourselves to solvable problems, like learning a programming language.

Within the formal world of Computer Science our description of an algorithm would be considered a little lax. For our introductory needs it is sufficient, but a more rigorous approach is given by Hopcroft and Ullman in *Introduction to Automata Theory, Languages and Computation*, and by Beckman in *Mathematical Foundations of Programming.*

Problems

1. What is an algorithm?

2. What distinguishes top-down from bottom-up approaches to problem solving? Illustrate your answer with reference to the problem of a car, motorcycle or bicycle having a flat tire.

Bibliography

Aho A. V., Hopcroft J. E., Ullman J. D., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1982.

Theoretical coverage of the design and analysis of computer algorithms.

Beckman F. S., *Mathematical Foundations of Programming*, Addison Wesley, 1981

Good clear coverage of the theoretical basis of computing.

Dahl O. J., Dijkstra E. W., Hoare C. A. R., *Structured Programming,* Academic Press, 1972.

This is the seminal book on structured programming.

Davis M., *Computability and Unsolvability,* Dover, 1982.

The book is an introduction to the theory of computability and non-computability – the theory of recursive functions in mathematics. Not for the mathematically faint hearted!

Davis W. S., *Systems Analysis and Design*, Addison Wesley, 1983.

Good introduction to systems analysis and design, with a variety of case studies. Also looks at some of the tools available to the systems analyst.

Fogelin R. J., *Wittgenstein*, Routledge and Kegan Paul, 1980.

The book provides a gentle introduction to the work of the philosopher Wittgenstein, who examined some of the philosophical problems associated with logic and reason.

Hopcroft J. E., Ullman J. D., *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979.

Comprehensive coverage of the theoretical basis of computing.

Kernighan B. W., Plauger P. J., *Software Tools*, Addison Wesley, 1976.

Interesting essays on the program development process, originally using a non-standard variant of Fortran. Also available using Pascal.

Knuth D. E., *The Art of Computer Programming*, Addison Wesley,

Vol 1. *Fundamental Algorithms*, 1974

Vol 2. *Semi-numerical Algorithms*, 1978

Vol 3. *Sorting and Searching*, 1972

Contains interesting insights into many aspects of algorithm design. Good source of specialist algorithms, and Knuth writes with obvious and infectious enthusiasm (and erudition).

Millington D., *Systems Analysis and Design for Computer Applications*, Ellis Horwood, 1981.

Short and readable introduction to systems analysis and design.

Wirth N., *Program Development by Stepwise Refinement*, Communications of the ACM, April 1971, Volume 14, Number 4, pp. 221-227.

Clear and simple exposition of the ideas of stepwise refinement.

3

Introduction to Programming Languages

We have to go to another language in order to think clearly about the problem.

Samual Delaney, Babel–17

‘Where shall I begin, please your Majesty?’ he asked
‘Begin at the beginning,’ the King said, gravely, ‘and go on till you come to the end: then stop.’

Lewis Carroll, Alice’s Adventures in Wonderland.

Aims

The primary aim of this chapter is to provide a short history of program language development. It concentrates on some but not all of the major milestones of the last 40 years, in rough chronological order. The secondary aim is to show the breadth of languages available. The chapter concludes with coverage of a small number of more specialised languages.

Introduction

It is important to realise that programming languages are a recent invention. They have been developed over a relatively short period – 40 years, and are still undergoing improvement. Time spent gaining some historical perspective will help you understand and evaluate future changes. This chapter starts right at the beginning and takes you through some, but not all, of the developments during this 40 year span. The bulk of the chapter restricts itself to languages that are reasonably widely available commercially, and therefore ones that you are likely to meet. The chapter concludes with a coverage of some more specialised and/or recent developments.

Some Early Theoretical Work

Some of the most important early theoretical work in computing was that of *Turing* and *von Neumann*. Turing's work provided the base from which it could be shown that it was possible to get a machine to solve problems. The work of von Neumann added the concept of storage and combined with Turing's work to provide the basis of most computers designed to this day.

What is a programming language ?

For a large number of people a programming language provides the means of getting a digital computer to solve a problem. There are a wide range of problems, and an equally wide range of programming languages, with particular programming languages being suited to a particular class of problems. Thus there are a wide variety of programming languages, which often appears bewildering to the beginner.

Program Language Development and Engineering

There is much in common between the development of programming languages and the development of anything from the engineering world. Consider the car: old cars offer much of the same functionality as modern ones, but most people prefer driving newer ones. The same is true of programming languages, where you can achieve much with the older languages, but the newer ones are easier to use.

The Early Days

A concept that proves very useful when discussing programming languages is that of the level of a machine. By this is meant how close a language is to the underlying machine that the program *runs* on. In the early days of programming (up to 1954) there were only two broad categories, machine languages and assemblers. The language that a digital machine uses is that of 0 and 1, i.e. they are binary devices. Writing a program in terms of patterns of 0 and 1 was not particularly satisfactory and the capability of using more meaningful mnemonics was soon introduced. Thus it was realised quite quickly that one of the most important aspects of programming languages is that they have to be read and understood by *both* machines and humans.

Fortran

The next stage was the development of higher level languages. The first of these was *Fortran* and it was developed over a three year period from 1954 to 1957 by an IBM team lead by John Backus. This group achieved considerable success, and helped to prove that the way forward lay with high level languages for computer based problem solving. Fortran stands for *for*mula *trans*lation and was used mainly by people with a scientific background for solving problems that had a significant arithmetic content. It was thus relatively easy, for the time, to express this kind of problem in Fortran.

By 1966 and the first standard Fortran was widely available, easy to teach, had demonstrated the benefits of subroutines and independent compilation, was relatively machine independent and often had very efficient implementations. Possibly the single most important fact about Fortran was and still is its widespread usage in the scientific community.

Cobol

The business world also realised that computers were useful and several languages were developed including FLOWMATIC, AIMACO, Commercial Translator and FACT, leading eventually to Cobol - *Co*mmon *B*usiness *O*rientated *L*anguage. There is a need in commercial programming to describe data in a much more complex fashion than for scientific programming, and thus Cobol had far greater capability in this area than Fortran. The language was unique at the time in that a group of competitors worked together with the objective of developing a language that would be useful on machines used by other manufacturers.

The contributions made by Cobol include

- firstly the separation between
    - the task to be undertaken
    - the description of the data involved
    - the working environment in which the task is carried out
- secondly a data description mechanism that was largely machine independent
- thirdly its effectiveness for handling large files

- fourthly the benefit to be gained from a programming language that was easy to read

Modern developments in computing, of report generators, file handling software, fourth generation development tools, and especially the increasing availability of commercial relational database management systems are gradually replacing the use of Cobol, except where high efficiency and or tight control are required.

Algol

Another important development of the 1950's was Algol. It had a history of development from Algol 58, the original Algol language, through Algol 60 eventually to the Revised Algol 60 Report. Some of the design criteria for Algol 58 were

- the language should be as close as possible to standard mathematical notation and should be *readable* with little further explanation
- it should be possible to use it for the description of computing processes in publications
- the new language should be mechanically translatable into machine programs

A sad feature of Algol 58 was the lack of any input/output facilities, and this meant that different implementations often had incompatible features in this area.

The next important step for Algol occurred at a UNESCO sponsored conference in June 1959. There was an open discussion on Algol and the outcome of this was Algol 60, and eventually the Revised Algol 60 Report.

It was at this conference that John Backus gave his now famous paper on a method for defining the syntax of a language, called Backus Normal Form, or BNF. The full significance of the paper by Backus was not immediately recognised. However BNF was to prove of enormous value in language definition, and helped provide an interface point with computational linguistics.

The contributions of Algol to program language development include

- block structure
- scope rules for variables because of block structure
- the BNF definition by Backus – most languages now have a formal definition
- the support of recursion
- its offspring

and thus Algol was to prove to make a contribution to programming languages that was never reflected in the use of Algol 60 itself, in that it has been the parent of one of the main strands of program language development.

Chomsky and Program Language Development

Programming languages are of considerable linguistic interest, and the work of Chomsky in 1956 in this area was to prove of inestimable value. Chomsky's system of transformational grammar was developed in order to give a precise mathematical description to certain aspects of language. Simplistically Chomsky describes grammars and these grammars in turn can be used to define or generate corresponding kinds of languages. It can be shown that for each type of grammar and language there is a corresponding type of machine. It was quickly realised that there was a link with the earlier work of Turing.

This link helped provide a firm scientific base for programming language development, and modern compiler writing has come a long way from the early work of Backus and his team at IBM. It may seem important when playing a video game at home or in an arcade, but for some it is very comforting that there is a firm theoretical basis behind all that fun.

Lisp

There were developments in very specialised areas also. List processing was proving to be of great interest in the 50's and saw the development of IPLV between 1954 and 1958. This in turn lead to the development of Lisp at the end of the 50's. It has proved to be of considerable use for programming in the areas of artificial intelligence, playing chess, automatic theorem proving and general problems solving. It was one of the first languages to be interpreted rather than compiled. Whilst interpreted languages are invariably slower and less efficient in their use of the underlying computer system, than compiled languages, they do provide great opportunities for the user to explore and try out ideas whilst sat at a terminal. The power that this gives to the computational problem solver is considerable.

Possibly the greatest contribution to program language development made by Lisp was its functional notation.

Snobol

Snobol was developed to aid in string processing which was seen as an important part of many computing tasks e.g. parsing of a program. Probably the most important thing that Snobol demonstrated was the power of pattern matching in a programming language, e.g. it is possible to define a pattern for a title that would include Mr, Mrs, Miss, Rev, etc and search for this pattern in a text using Snobol. Like Lisp it is generally available as an interpreter rather than a compiler, but compiled versions do exist, and are often called Spitbol. Pattern matching capabilities are now to be found in many editors and this makes them very powerful and useful tools. It is in the area of pattern matching that Snobol's greatest contribution to program language development lies.

Second Generation Languages

PL/1 and Algol 68

It is probably true that Fortran, Algol 60 and Cobol are the three main first generation high level languages. The 60's saw the emergence of PL/1 and Algol 68. PL/1 was a synthesis of features of Fortran, Algol 60 and Cobol. It was soon realised that whilst PL/1 had great richness and power of expression this was in some ways offset by the greater difficulties involved in language definition and use.

These latter problems were true of Algol 68 also. The report introduced its own syntactic and semantic conventions and thus forced upon the prospective user another stage in the learning process. However it has a small but very committed user population who like the very rich facilities provided by the language.

Simula

Another strand that makes up program language development is provided by Simula. It is a general purpose programming language developed by Dahl, Myhrhaug and Nygaard of the Norwegian Computing Centre. The most important contribution that Simula makes is the provision of language constructs that aid the programming of complex, highly interactive problems. It is thus heavily used in the areas of simulation and modelling.

Pascal

The designer of Pascal, Niklaus Wirth, had participated in the early stages of the design of Algol 68 but considered that the generality and complexity of Algol 68 was a move in the wrong direction. Pascal like Algol 68 had its roots in Algol 60 but aimed at providing expressive power through a small set of straightforward concepts. This set is relatively easy to learn and helps in producing readable and hence more comprehensible programs.

APL

APL is another interesting language of the 60's. It was developed by Iverson in the early 1960's and was available by the mid to late 60's. It is an interpretive vector and matrix based language with an extensive set of operators for the manipulation of vectors, arrays etc of whatever data type. As with Algol 68 it has a small but dedicated user population. A possibly unfair comment about APL programs is that you do not debug them, but rewrite them!

Basic

Basic stands for *B*eginners *A*ll Purpose *S*ymbolic *I*nstruction *C*ode, and was developed by Kemeny and Kurtz at Dartmouth during the 1960's. Its name gives a clue to its audience and it is very easy to learn. It is generally interpreted, though compiled versions do exist. It is probably the most heavily used language on micros and home computers. It has proved to be well suited to the rapid development of small programs. It is much criticised because it lacks features that encourage or force the adoption of sound programming techniques.

C

There is a requirement in computing to be able to access directly or at least efficiently the underlying machine. It is therefore not surprising that computer professionals have developed high level languages to do this. This may well seem a contradiction, but it can be done to quite a surprising degree. Some of the earliest published work was that of Martin Richards and the development of BCPL.

This language directly influenced the work of Ken Thompson and can be clearly seen in the programming languages B and C. The UNIX operating system is almost totally written in C and demonstrates very clearly the benefits of the use of high level languages wherever possible.

Some Other Strands in Language Development

There are many strands that make up program language development and some of them are introduced here.

Abstraction, Stepwise Refinement and Modules

Abstraction has proved to be very important in programming. It enables a complex task to be broken down into smaller parts concentrating on *what* we want to happen rather than *how* we want it to happen This leads almost automatically to the ideas of stepwise refinement and modules, with collections of modules to perform specific tasks or steps.

Structured Programming

Structured programming in its narrowest sense concerns itself with the development of programs using a small but sufficient set of statements and in particular control statements. It has had a great effect on program language design and most languages now support the minimal set of control structures.

In a broader sense structured programming subsumes other objectives including simplicity, comprehensibility, verifiability, modifiability and maintenance of programs.

Ada

Ada represents the culmination of many years of work in program language development. It was a collective effort and the main aim was to produce a language suitable for programming large scale and real time systems. Work started in 1974 with the formulation of a series of documents by the American Department of Defence (DoD), which lead to the Steelman documents. It is a modern algorithmic language with the usual control structures, and facilities for the use of modules and allows separate compilation with type checking across modules.

Ada is a powerful and well engineered language. Its widespread use is certain as it has the backing of the DoD. However it is a large and complex language and consequently requires some effort to learn. It seems unlikely to be widely used except by a small number of computer professionals.

Modula

Modula was designed by Wirth during the 1970's at ETH, for the programming of embedded real time systems. It has many of the features of Pascal, and can be taken for Pascal at a glance. The key new features that Modula introduced were those of processes and monitors.

As with Pascal it is relatively easy to learn and this makes it much more attractive than Ada for most people, achieving much of the capability, without the complexity.

Modula 2

Wirth carried on developing his ideas about programming languages and the culmination of this can be seen in Modula 2, and in his words

...In 1977, a research project with the goal to design a computer system (hardware and software) in an integrated approach, was launched at the Institut fur Informatik of ETH Zurich. This system (later to be called Lilith) was to be programmed in a single high level language, which therefore had to satisfy requirements of high level system design as well as those of low level programming of parts that closely interact with the given hardware. Modula 2 emerged from careful design deliberations as a language that includes all aspects of Pascal and extends them with the important module concept and those of multiprogramming. Since its syntax was more in line with Modula than Pascal's the chosen name was Modula 2. ...

The language's main additions with regard to Pascal are:

1. the module concept, and in particular the facility to split a module into a definition part and an implementation part.

2. A more systematic syntax which facilitates the learning process. In particular, every structure starting with a keyword also ends with a keyword, i.e. is properly bracketed.

3. The concept of process as the key to multi-programming facilities.

4. So called low level facilities which make it possible to breach the rigid type consistency rules and allow to map data with Modula 2 structure onto a store without inherent structure.

5. The procedure type which allows procedures to be dynamically assigned to variables.

Other Language Developments

The following is a small selection of language developments that the author finds interesting – they may well not be included in other peoples coverage.

Logo

Logo is a language that was developed by Papert and colleagues at the Artificial Intelligence Laboratory at MIT. Papert is a professor of both mathematics and education, and has been much influenced by the psychologist Piaget. The language is used to create learning environments in which children can communicate with a computer. The language is primarily used to demonstrate and help children develop fundamental concepts of mathematics. Probably the *turtle* and *turtle geometry* are known by educationalists outside of the context of Logo. Turtles have been incorporated into the Smalltalk computer system developed at Xerox Palo Alto Research Centre – Xerox PARC.

Postscript, TeX

The 80's have seen a rapid spread in the use of computers for the production of printed material. The above languages are each used in this area quite extensively.

Postscript is a low level interpretive programming language with good graphics capabilities. Its primary purpose is to enable the easy production of pages containing text, graphical shapes and images. It is rarely seen by most end users of modern desk top publishing systems, but underlies many of these systems. It is supported by an increasing number of laser printers and type-setters.

TeX is a language designed for the production of mathematical texts, and was developed by Donald Knuth. It linearises the production of mathematics using a standard computer keyboard. It is widely used in the scientific community for the production of documents involving mathematical equations.

Prolog

Prolog was originally developed at Marseille by a group lead by Colmerauer in 1972/73. It has since been extended and developed by a variety of people including Pereira (L.M.), Pereira (F), Warren and Kowalski. Prolog is unusual in that it is a vehicle for *logic* programming. Most of the languages described here are basically algorithmic languages and require a specification of *how* you want something done. Logic programming concentrates on the *what* rather than the *how*. The language appears strange at first, but has been taught by Kowalski and others to 10 year old children at schools in London.

Smalltalk

Smalltalk has been under development by the Xerox PARC Learning Research Group since the 1970's. In their words *Smalltalk is a graphical, interactive programming environment. As suggested by the personal computer vision, Smalltalk is designed so that every component in the system is accessible to the user and can be presented in a meaningful way for observation and manipulation. The user interface issues in Smalltalk revolve around the attempt to create a visual language for each object. The preferred hardware system for Smalltalk includes a high resolution graphical display screen and a pointing device such as a graphics pen or mouse. With these devices the user can select information viewed on the screen and invoke messages in order to interact with the information.* Thus Smalltalk represents a very different strand in program language development. The ease of use of a system like this has long been appreciated and was first demonstrated commercially by the Macintosh micro-computers.

Wirth has spent some time at Xerox PARC and has been influenced by their work, and in his own words *the most elating sensation was that after sixteen years of working for computers the computer now seemed to work for me*. This influence can be seen in the design of the Lilith machine, the Modula2 engine.

SQL

SQL stands for Structured Query Language, and was originally developed by a variety of people mainly working for IBM in the San Jose Research Laboratory. It is a relational database language, and enables programmers to define, manipulate and control data in a relational database. Simplistically a relational database is seen by a user as a collection of tables, comprising rows and columns. It has become the most important language in the whole database field.

ICON

Icon is in the same family as Snobol, and is a high-level general purpose programming language that has most of the features necessary for efficient processing of non-numeric data. Griswold (one of the original design team for

Snobol) has learnt much since the design and implementation of Snobol, and the language is a joy to use in most areas of text manipulation.

Fortran 8x

Almost as soon as the Fortran 77 standard was complete and published work began on the next version. At the moment the 8x standard is only a draft, but should be completed very soon. The language draws on many of the ideas covered in this chapter and these help to make Fortran 8x a very promising language for the future. Some of the new features include

user defined data types

array operations

control of the precision of numerical computation

enhanced control structures

recursion

dynamic storage allocation

A very readable coverage of the new standard can be found in *Fortran 8x Explained* by Metcalfe and Reid. It is likely that 8x conformant compilers will become available in the near future.

Fortran 77 Revisited

As should now be apparent Fortran is but one of a large family of programming languages. It has already been standardised twice (in 1966 and 1978), and is nearing completion of the third standardisation exercise to produce Fortran 8x. The X3J3 committee was too circumspect to have called it Fortran 88, after missing Fortran 77 by a year, though they look like missing this deadline now! All of Fortran 77 is contained in 8x, so don’t worry about wasting your time learning Fortran 77 – it will be around for a long time to come!

Summary

It is hoped that the reader now has some idea about the wide variety of uses that programming languages are put to. Do not be put off by the range of languages described here. All journeys have to start somewhere, and on the journey of mastery of programming Fortran is a good place to start!

Bibliography

Adobe Systems Incorporated, *Postscript Language Tutorial and Cookbook*, Addison Wesley.

Adobe Systems Incorporated, *Postscript Language Reference Manual*, Addison Wesley.

> The two books provide a comprehensive coverage of the facilities and capabilities of Postscript.

Annals of the History of Computing, *Special Issue: Fortran's 25 Anniversary,* ACM publication.

> Very interesting comments, some anecdotal, about the early work on Fortran.

Birtwistle G.M., Dahl O. J., Myhrhaug B., Nygaard K., *SIMULA BEGIN*, Chartwell-Bratt Ltd.

> A number of chapters in the book will be of interest to programmers unfamiliar with some of the ideas involved in a variety of areas including systems and models, simulation, and co-routines. Also has some sound practical advice on problem solving.

Brinch-Hansen P., *The Programming Language Concurrent Pascal,* IEEE Transactions on Software Engineering, June 1975, 199-207.

> Looks at the extensions to Pascal necessary to support concurrent processes.

Date C. J., *A Guide to the SQL Standard*, Addison Wesley.

> Date has written extensively on the whole database field, and this book looks at the SQL language itself. As with many of Dates works quite easy to read. Appendix F provides a useful SQL bibliography.

Geissman L. B., *Separate Compilation in Modula2 and the structure of the Modula2 Compiler on the Personal Computer Lilith*, Dissertation 7286, ETH Zurich

Jacobi C., *Code Generation and the Lilith Architecture,* Dissertation 7195, ETH Zurich

> Fascinating background reading concerning Modula2 and the Lilith architecture.

Goldberg A., and Robson D., *Smalltalk 80: The language and its implementation,* Addison Wesley.

> Written by some of the Xerox PARC people who have been involved with the development of Smalltalk. Provides a good

introduction (if that is possible with the written word) of the capabilities of Smalltalk.

Goos and Hartmanis (Eds), *The Programming Language Ada - Reference Manual*, Springer Verlag.

The definition of the language.

Griswold R. E., Poage J. F., Polonsky I. P., *The Snobol4 Programming Language*, Prentice-Hall.

The original book on the language. Also provides some short historical material on the language.

Griswold R. E., Griswold M. T., *The Icon Programming Language*, Prentice-Hall.

The definition of the language with a lot of good examples. Also contains information on how to obtain public domain versions of the language for a variety of machines and operating systems.

Hoare C.A.R., *Hints on Programming Language Design,* SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1973.

The first sentence of the introduction sums it up beautifully: *I would like in this paper to present a philosophy of the design and evaluation of programming languages which I have adopted and developed over a number of years, namely that the primary purpose of a programming language is to help the programmer in the practice of his art.*

Jenson K., Wirth N., *Pascal: User Manual and Report*, Springer Verlag.

The original definition of the Pascal language. Understandably dated when one looks at more recent expositions on programming in Pascal.

Kemeny J.G., Kurtz T.E., *Basic Programming,* Wiley.

The original book on Basic by its designers.

Kernighan B. W., Ritchie D. M., *The C Programming Language,* Prentice Hall: Englewood Cliffs, New Jersey.

The original work on the C language, and thus essential for serious work with C.

Kowalski R., *Logic Programming in the Fifth Generation*, The Knowledge Engineering Review, The BCS Specialist Group on Expert Systems.

A short paper providing a good background to Prolog and logic programming, with an extensive bibliography.

Knuth D. E., *The TeXbook*, Addison Wesley.

> Knuth writes with an tremendous enthusiasm and perhaps this is understandable as he did design TeX. Has to be read from cover to cover for a full understanding of the capability of TeX.

Lyons J., *Chomsky,* Fontana/Collins, 1982.

> A good introduction to the work of Chomsky, with the added benefit that Chomsky himself read and commented on it for Lyons. Very readable.

Malpas J., *Prolog: A Relational Language and its Applications,* Prentice-Hall.

> A good introduction to Prolog for people with some programming background. Good bibliography. Looks at a variety of versions of Prolog.

Marcus C., *Prolog Programming: Applications for Database Systems, Expert Systems and Natural Language Systems*, Addison Wesley.

> Coverage of the use of Prolog in the above areas. As with the previous book aimed mainly at programmers, and hence not suitable as an introduction to Prolog as only two chapters are devoted to introducing Prolog.

Metcalf M. and Reid J., *Fortran 8x Explained*, Oxford Science Publications, OUP.

> A clear compact coverage of the main features of Fortran 8x. Reid is secretary of the X3J3 committee.

Papert S., *Mindstorms - Children, Computers and Powerful Ideas,* Harvester Press

> Very personal vision of the uses of computers by children. It challenges many conventional ideas in this area.

Sammett J., *Programming Languages: History and Fundamentals,* Prentice Hall.

> Possibly the most comprehensive introduction to the history of program language development – ends unfortunately before the 1980's.

Sethi Ravi, *Programming Languages: Concepts and Constructs*, Addison Wesley.

> Eminently readable and thorough coverage of programming languages. The annotated bibliographic notes at the end of each chapter and the extensive bibliography make it a very useful book.

Young S. J., *An Introduction to Ada, 2nd Edition*, Ellis Horwood.

> A readable introduction to Ada. Greater clarity than the first edition.

Wirth N., *An Assessment of the Programming Language Pascal*, IEEE Transactions on Software Engineering, June 1975, 192-198.

Wirth N., *History and Goals of Modula2*, Byte, August 1984, 145-152.

> Straight from the horse's mouth!

Wirth N., *On the Design of Programming Languages*, Proc. IFIP Congress 74, 386-393, North Holland, Amsterdam.

Wirth N., *The Programming Language Pascal*, Acta Informatica 1, 35-63, 1971.

Wirth N., *Modula: a language for modular multi- programming*, Software Practice and Experience, 7, 3-35, 1977.

Wirth N., *Programming in Modula2*, Springer Verlag.

> The original definition of the language. Essential reading for anyone considering programming in Modula2 on a long term basis.

4

Introduction to Using a Computer System

Plug in! Playback! Tapespond! The electronic network longs to set you free.

Edwin Morgan, 'Schools's Out'

Aims

The aims of this chapter are to introduce a small set of concepts to enable you to work at a computer system, eithor personal workstation or timesharing system. In particular:–

- the overall environment that programming is carried out in
- operating systems
- files
- editors
- the compilation process
- linking and libraries

Introduction

Programming involves the use of a computer system, and therefore to become successful as a programmer you are required to learn how to use a computer system effectively. In particular you need to know about

- operating systems
- files
- editors and editing
- the compilation process
- linking

Operating Systems

A simple definition of an operating system is the suite of programs that make the hardware usable. Most computer systems have an operating system and they vary considerably from those available on micros, e.g. CP/M on 8 bit systems, PCDOS/MSDOS/DOS on 8, 16 and 32 bit machines respectively, through VAX/VMS on DEC mini computers, and VM/CMS on IBMs large general purpose mainframes.

From the programmer viewpoint the operating system must provide ways of

- creating, editing and deleting files
- copying files
- compiling, linking and running programs
- saving files
- get printed versions of files

Files

A file is a collection of information that you refer to by name, e.g. if you were to use a word processor to prepare a letter then the letter would exist as a file on that system, generally on a disk of some sort. In a database containing information on student examination results and course-work marks, this information would exist as a file.

There will be many ways of manipulating files on the computer system that you work on. The interface to the computer system is provided by an operating system, and unfortunately most operating systems offer similar functionality, via commands that sometimes have similar names, but different syntax, or completely different names. So when we look at ways of manipulating files we will generally have to learn several ways of doing the same thing, if we use more than one computer system. You will use an editor or word processor to make changes to your program. This file would be the source of your program, and you would then use a compiler to compile you program. The compiler will probably generate a number of files as it compiles your program, some for its own internal use, others that may well be of interest to you. There will be commands to make files permanent, commands to copy files, commands to store data on magnetic tape or floppy disks, etc. Some files can be read by human beings e.g. text files like the source of your program, others can only be understood by the machine, so called binary files that are the actual bits patterns that the machine itself understands. Therefore it is not always possible to examine all files on a computer system, as some are not intended to be immediately or directly comprehensible by human beings.

You will have to learn about

- what files belong to you;
- how to get rid of files;
- to get on–line help;
- getting files printed;
- display a file on the screen

and the following table looks at three operating systems (MSDOS, VAX/VMS and UNIX) and the terminology or commands they use to achieve the above.

Operating system –>	MSDOS	UNIX	VAX/VMS
What files are mine	dir	ls	dir
Getting rid of files	del/era	rm	del/pur
On-line help		man	help
Printing files	print	pr	pr
Display a file	type	cat	type

Editors and Editing

All general purpose computer systems have at least one editor so that you can create and modify text files like your program and data files. The easiest editor for most people to use is a screen editor. All this means is that you can move a cursor around the screen of your terminal or work-station and make changes to your file by typing in the new or modified text. The changes take place immediately as you type in the text. Screen editors are the most widely used type of editor.

When you use a context editor you have to type in commands to change the text, e.g.

```
s/Fred/Bert/
```

would take the first occurrence of Fred and substitute it with Bert. These editors are easy to use since you only need a small number of commands to do most of what you want.

There are also editors with pattern matching capabilities. This means that you can define a pattern, and then search for any of the possible strings that that pattern represents, e.g.

```
[0..9][0..9][0..9]
```

is a possible pattern for a three digit number, each component

```
[0..9]
```

meaning any digit between 0 and 9. Therefore

```
s/[0..9][0..9][0..9]//
```

would remove the next three digit number occurring within a line, or put another way substitute it with nothing.

On micro systems the editor may well be called a word processor, and there will be a way of using the word processor in program mode, rather than word processing mode. In word processing mode you will probably find that the text will justify automatically left and right, whereas in program mode the text remains as you have typed it in.

Editors that are program language sensitive, and help in consistent layout, i.e. indentation, and style, are now becoming more widely available, and help the beginner considerably in learning how to use a programming language effectively.

The Compilation Process

It is here that we *feed* our program prepared using an editor or word processor into a compiler. If the term seems a little farfetched wait until you have used a compiler a number of times and seen the sort of (apparent) gibberish that can be generated from a very small error. When we eventually get a program to compile we then move onto the next stage, using a linker.

Linking

A variety of names are given by different computer manufacturers to the linker, including linker, link loader, linkage editor and loader. The underlying functionality is the same and they are used in the middle part of the compile, link and run or execute cycle. They take the output from the compiler and turn it into something that can be executed or run on the computer.

Running a Program

It is at this stage that we actually see whether our program works. If we are lucky it will execute as we intended and we have solved our problem. We typically interact with it and find that it doesn't do what we want, and we then

- locate the error
- edit the program
- compile the program
- link the program
- run it, find it doesn't work.....

and as can be seen there is a loop that we cycle through until we have a working program.

We will look in more detail at the compilation process in the chapters on functions, subroutines, common and data.

Bibliography

The best strategy here is to obtain the manufacturers documentation for the system you use. Each manufacturer will have a particular style and terminology, and whilst one manufacturer may use consistent and clear terminology (I live in hope!) you will inevitably find that different manufacturers use contradictory terminology.

Deitel H., *Operating Systems*, Addison Wesley, 1984

> The book provides a comprehensive coverage of most aspects of operating systems. There are also a number of case studies of current operating systems.

Lister A. M., *Fundamentals of Operating Systems*, MacMillan, 1984.

> Brief and straightforward coverage of essential aspects of operating systems.

5

Introduction to Programming

'Though this be madness, yet there is method in't'

Shakespeare

Aims

The aims of the chapter are:–

- to introduce the idea that there is a wide class of problems that may be solved with a computer, and that there is a relationship between the kind of problem to be solved and the choice of programming language that is used to solve that problem;
- to give some ofthe reasons for the choice of Fortran 77
- to introduce the fundamental components or kinds of statements to be found in a general purpose programminglanguage
- to introduce the three concepts of name, type and value

Introduction

We have seen that an algorithm is a sequence of steps that will solve a part or the whole of a problem. A program is the realisation of an algorithm in a programming language, and there are at first sight a surprisingly large number of programming languages. The reason for this is that there are a wide range of problems that are solved using a computer, e.g. the telephone company generating a telephone directory or the meteorological centre producing a weather forecast. These two problems make different demands on a programming language, and it is unlikely that the same language would be used to solve both.

The range of problems that you want to solve will therefore have a strong influence on the programming language that you use. FORTRAN stands for FORmula TRANslation, which gives a hint of the expected range of problems. Fortran is particularly *good* at numerical problems. It is designed to do things with numbers, and to calculate. It can manipulate character information (a feature absent in earlier versions of the language). These two features make Fortran a good all-round language. Some of the reasons for choosing Fortran are:–

- The language is suitable for a wide class of both numeric and non-numeric problems;
- The language is widely available in both the educational and scientific sectors;
- A lot of software already exists, written in either Fortran 77 or its, predecessor, Fortran 66. The numbers are based on the year of the definition of the standard, 1966 or 1977. Fortran 66 is also known as Fortran IV.

Periodically, Fortran is *re-defined* by the American Standards Institute, who have a committee — the X3J3 committee — which considers changes to the language. This new definition is then published as the definitive statement of the form of the language. Although these changes take place about every ten years, we are confident that we will be using Fortran well into the next century. Therefore there will be both short-term and long-term benefits from learning Fortran 77. In the following, Fortran is taken to mean Fortran 77.

Elements of a programming language

As with ordinary (so-called *natural*) languages, e.g. English, French, Gaelic etc., programming languages have rules of syntax, grammar and spelling. The application of the rules of syntax, grammar and spelling in a programming language are more strict. A program has to be unambiguous, since it is a *precise* statement of the actions to be taken. Many everyday activities are rather vaguely defined — *Buy some bread on your way home* — but we are generally sufficiently adaptable to cope with the variations which occur as a result. If, in a program to calculate wages, we had an instruction *Deduct some money* for tax and insurance we could have an awkward problem when the program calculated completely different wages for the same person for the same amount of work every time it was run. One of the implications of the strict syntax of a programming language for the novice is that apparently silly error messages will appear when first writing programs. As with many other *new* subjects you will have to learn some of the jargon to understand these messages.

Programming languages are made up of statements. These statements fall into the following broad categories:–

- **Data description statements**

  These are necessary to describe what kinds of data are to be processed. In the wages program for example, there is obviously a difference between peoples names and the amount of money they earn, i.e. these two things are not the same, and it would not make any sense adding your name to your wages. The technical term for this is *data type;* a wage would be of a different data type (a number) to a surname (a sequence of characters).

- **Control structures**

  A program can be regarded as a sequence of statements to solve a particular problem, and it is common to find that this sequence needs to be varied in practice. Consider again the wages program. It will need to select between a variety of circumstances (say married or single, paid weekly or monthly etc), and also to repeat the program for everybody employed. So there is the need in a programming language for statements to vary and/or repeat a sequence of statements.

- **Data processing statements**

  It is necessary in a programming language to be able to process data. The kind of processing required will depend on the kind or type of data. In the wages program, for example, you will need to distinguish between names and wages. Therefore there must be different kinds of statements to manipulate the different types of data, i.e. *wages* and *names*.

- **Input and output (i/o) statements**

  For flexibility, programs are generally written so that the data that they work on exists *outside* the program. In the wages example the details for each person employed would exist in a *file* somewhere, and there would be a *record* for each person in this file. This means that the program would not have to be modified each time a person left, was ill etc., although the individual records might be updated. It is easier to modify data than to modify a program, and less likely to produce unexpected results. To be able to vary the action there must be some mechanism in a programming language for getting the data *into* and *out of* the program. This is done using input and output statements, sometimes shortened to i/o statements.

Let us now consider a simple program which will read in somebody's name and print it out.

```
      PROGRAM INOUT
C
C This program reads in and prints out a name
C
      CHARACTER NAME*20
      PRINT *,' Type in your name, up to 20 characters'
      PRINT *,' enclosed in quotes'
      READ *,NAME
      PRINT *,NAME
      END
```

There are several very important points to be covered here, and they will be taken in turn:–

- Each line is a statement.
- There is a sequence to the statements. The statements will be processed in the order that they are presented, so in this example the sequence is *print, print, read, print.*
- Statements start in column 7. There are exceptions to this rule, as we will see later, but, nevertheless, the main part of any statement must lie between columns 7 and 72. This is an inherited feature which goes back to the days when communication with a computer was through 80-column cards. The last eight columns were often used for sequencing information (just in case you dropped your cards!). For terminal use, having to start in column 7 can be irritating; but since you are using a terminal, there will probably be *tab* commands you can use, so that you skip over the first few columns by pressing a single tab key.
- The first statement names the program. It makes sense to choose a name that conveys something about the purpose of the program.
- The next three lines are *comment* statements. They are identified by a *C* in the first column. This is your first exception to the column 7 rule above. Comments are inserted in a program to explain the purpose of the program. They should be regarded as an

integral part of all programs. It is essential to get into the habit of inserting comments into your programs straight away.

- The CHARACTER statement is a *type* declaration. It was mentioned earlier that there are different kinds of data. There must be some way of telling the programming language that this data is of a certain type, and that therefore certain kinds of operation are allowed and others are banned or just plain stupid! It would not make sense to add a name to a number, e.g. what does Fred+10 mean? So this statement defines that the *variable* NAME to be of type CHARACTER and only character operations are permitted. The concept of a variable is covered in the next section.
- The PRINT statements print out an informative message to the terminal – in this case a guide as to what to type in. The use of informative messages like this throughout your programs is strongly recommended.
- The READ statement is one of the i/o statements. It is an instruction to *read* from the terminal or keyboard; whatever is typed in from the terminal will end up being associated with the variable NAME. Input/output statements will be explained in greater detail in later sections.
- The PRINT statement is another i/o statement. This statement will print out what is associated with the variable NAME and in this case, what you typed in.
- The END statement terminates this program. It can be thought of as being similar to a full stop in natural language, in that it finishes the program, in the same way that a . ends a sentence.
- Note also the use of the '*' in three different contexts.
- Lastly, when you do run this program, you must put a prime or apostrophe (') before the characters you input, and another prime or apostrophe after them. These apostrophes will not be printed out by the program. This feature is a result of using READ * to input the characters, and later we will see ways of avoiding the use of apostrophes.

The above program illustrates the use of some of the statements in the Fortran language. Let us consider the action of the READ * statement in more detail. In particular, what is meant by a variable and a value.

Variables — name, type and value

The idea of a variable is one that you are likely to have met before, probably in a mathematical context. Consider the following tax payable = gross wages – tax allowances * tax rate

This is a simplified equation for the calculation of wage deductions. Each of the *variables* on the right hand side takes on *a value* for each person, which allows the calculation of the deductions for that person. The above equation expressed in Fortran would be an example of an *arithmetic assignment statement*. There is some arithmetic calculation taking place which yields a value, and this value is then assigned to the variable on the left hand side. This could be expressed in Fortran as

```
TAX= (GROSS–TAXALL) * TAXRAT
```

Note here the shortened form of the variable names. This is due to one of the *rules* of Fortran — a variable name must be six alphanumeric characters (letters and numbers) or less in length, and the first character must be a letter. A program typically consists of data processing statements that involve variables. It is a good idea to choose variable names that convey something meaningful about the use of that variable. Occasionally it is difficult to find meaningful names of only six characters, and it is not always easy to see what a variable is used for. In the problems, try to choose variable names that convey something meaningful about the use of the variable.

The following arithmetic assignment statement illustrates clearly the concepts of name and value, and the difference between the = in mathematics and computing:–

```
I=I+1
```

In Fortran this reads as take the current value of the variable *I* and add one to it, store the new value back into the variable *I*, i.e. *I* takes the value *I+1*. Algebraically,

$$i=i+1$$

does not make any sense.

Variables can be of different types, and the following show some of those available in Fortran.

Variable name	Data type	Value stored
GROSS	REAL	9440.30
TAXALL	INTEGER	2042
TAXRAT	REAL	.7
NAME	CHARACTER	ARTHUR

The concept of data type seems a little strange at first, especially as we commonly think of integers and reals as numbers. However, the benefits to be gained from this distinction are considerable. This will become apparent after writing several programs.

Let us consider another program now. This program reads in three numbers, adds them up and prints out both the total and the average.

```
      PROGRAM AVERAG
C
C THIS PROGRAM READS IN THREE NUMBERS AND SUMS
C AND AVERAGES THEM.
C
      REAL NUMBR1,NUMBR2,NUMBR3,AVRAGE,TOTAL
      INTEGER N
      N = 3
      TOTAL = 0.0
      PRINT *,'TYPE IN THREE NUMBERS'
      PRINT *,'SEPARATED BY SPACES OR COMMAS'
      READ *,NUMBR1,NUMBR2,NUMBR3
      TOTAL= NUMBR1+NUMBR2+NUMBR3
      AVRAGE=TOTAL/N
      PRINT *,'TOTAL OF NUMBERS IS',TOTAL
      PRINT *,'AVERAGE OF THE NUMBERS IS',AVRAGE
      END
```

Notes

- The program has been given a name that means something. As Fortran only allows a six character name, AVERAGE, with 7 letters, could not have been used.
- There are comments at the start of the program describing what the program does.
- The next two statements are type declarations. They define the variables to be of *real* or *integer* type. Remember integers are *whole* numbers, while real numbers are those which have a *decimal point.* For example, 2 is an integer, while 2.7, 2.00000001, and 2.0 are all real numbers. One of the fundamental distinctions in Fortran is between integers and reals. Type declarations must always come at the start of a program, before any *processing* is done.
- The first PRINT statement makes a text message, (in this case what is between the apostrophes) appear at the terminal. As was stated earlier it is good practice to put out a message like this so that you have some idea of what you are supposed to type in.

- The READ statement looks at the input from the keyboard (i.e. what you type) and in this instance associates these values with the three variables. These values can be separated by commas (,), spaces ( ), or even by pressing the carriage return key, i.e. they can appear on separate lines.
- The next statement actually does some data processing. It adds up the *values* of the three variables (NUMBR1, NUMBR2, and NUMBR3), and assigns the result to the variable TOTAL. This statement is called an *arithmetic assignment statement*, and is covered more fully in the next chapter.
- The next statement is another data processing statement. It calculates the average of the numbers entered and assigns the result to AVRAGE. We could have actually used the value 3 here instead, i.e. written AVRAGE=TOTAL/3 and had exactly the same effect. This would also have avoided the type declaration for NUMBER. However the original example follows established programming practice of declaring all variables and establishing their meaning unambiguously. We will see further examples of this type throughout the book.
- Finally the sum and average are printed out with suitable captions or headings. **Do not** write programs without putting captions on the results. It is too easy to make mistakes when you do this, or even forget what each number means.

Default variable types

The example used variables which held real numbers and integers. These are declared explicitly in the *declaration* statements at the beginning of the program. In general, we recommend the use of such *explicit* typing, where you declare your variables and their characteristics. However, you may also use *default* typing, where the initial letter of the variable name indicates its type – real or integer. The initial letter I, J, K, L, M or N is used to identify integer variables. Any other letter identifies a real variable. Thus, if you choose to employ default typing, and not override it by an explicit reference, X1, QUAD, HALF and ONE are all real variables, while NINE, INVERT, KOUNT and L2 are integers. Note that there is no default for character type variables; character variables *must* be declared explicitly.

Some more Fortran rules

There are certain things to learn about Fortran which have little immediate meaning, and some which have no logical justification at all, other than historical precedence. Why is a cat called a cat? At the end of some of the chapters there will be a brief summary of these *rules* or regulations when necessary. Here are a few:-

- There is an order to the statements in Fortran. Within the context of what you have covered so far, the order is:-
    - PROGRAM statement
    - Type declarations, e.g. INTEGER, REAL or CHARACTER
    - Processing and i/o statements
    - END statement
- Comments may appear anywhere in the program, after PROGRAM and before END, and must have a C or * in the very first column.

Fortran Character set

The Fortran character set comprises the following characters:-

Capital A through Z

The digits 0 through 9

and the characters:-

= equal

+ plus

– minus

* asterisk

/ slash or oblique

(left brackets (parenthesis)

) right brackets (parenthesis)

, comma

. decimal point

$ currency symbol

’ apostrophe

: colon

  blank (difficult to see as a character!)

Problems

1. Write a program that will read in your name and address and print them out in reverse order.

2. Type in the program AVERAG, given in this chapter. Demonstrate that the input may be separated by spaces or commas. Do you need the decimal point? What happens when you type in too much data? What happens when you type in too little?

6

Arithmetic

Taking Three as the subject to reason about —
A convenient number to state —
We add Seven, and Ten, and then multiply out
By One Thousand diminished by Eight.

The result we proceed to divide, as you see,
By Nine Hundred and Ninety and Two:
Then subtract Seventeen, and the answer must be
Exactly and perfectly true.

Lewis Carroll, 'The Hunting of the Snark'

Aims

The aims of this chapter are to introduce:–

- the rules for the evaluation of an arithmetic expression
- the idea of truncation and rounding of a number, and the care that must be taken
- to ensure that an arithmetic expression is evaluated as you intend
- the use of the PARAMETER statement to define or set up constants
- the idea that numbers on a computer have a finite size and precision

Arithmetic

Most problems in the educational and scientific community require arithmetic evaluation as part of the algorithm. As the rules for the evaluation of arithmetic in Fortran may differ from those that you are probably familiar with, you need to learn the Fortran rules thoroughly. In the previous chapter, we introduced the arithmetic assignment statement, emphasising the concepts of name, type and value. Here we will consider the way that arithmetic expressions are evaluated in Fortran.

The following are the five arithmetic operators available in Fortran:–

Mathematical operation	Fortran symbol or operator
Addition	+
Subtraction	–
Division	/
Multiplication	*
Exponentiation	**

Exponentiation is raising to a power. Note that the exponentiation operator is the * character *twice.*

The following are some examples of valid arithmetic statements in Fortran:–

```
TAX = GROSS – DEDUCT
COST = BILL + VAT +SERVIC
DELTA = DELTAX/DELTAY
AREA = PI * RADIUS * RADIUS
CUBE = BIG ** 3
```

The above expressions are all simple, and there are no problems when it comes to evaluating them. However, now consider the following:–

```
NETT = GROSS – ALLOW * TAXRAT
```

This is ambiguous. There is a choice of doing the subtraction before or after the multiplication. Experience with a calculator says that the subtraction would be done before the multiplication. However, if this expression was evaluated in Fortran the *multiplication* would be done before the subtraction.

Here are the rules for the evaluation of expressions in Fortran:–

- brackets are used to define priority in evaluation

- operators have a hierarchy of priority – a precedence. The hierarchy of operators is:–
  - **exponentiation;** when the expression has multiple exponentiation, they are evaluated right to left. For example,

    L=I**J**K

    is evaluated by first raising J to the power K, and then using this result as the exponent for I; more explicitly therefore

    L=I**(J**K)

    Although this is similar to the way in which we might expect an algebraic expression to be evaluated, it is not consistent with the rules for multiplication and division, and may lead to some confusion. When in doubt, use brackets.
  - **multiplication and division**; within successive multiplications and divisions, the order of evaluation is left to right. For example

    A=B*C/D*E

    would result in B and C being multiplied together; this result divided by D; and lastly the result of the division being multiplied by E.
  - **addition and subtraction**; as with multiplication and division, evaluation is carried on from left to right. However, it is seldom that the order of addition and subtraction is important, unless other operators are involved.

The following are all examples of valid arithmetic expressions in Fortran:–

```
SLOPE = (Y1-Y2)/(X1-X2)
X1=(-B+((B*B-4*A*C)**0.5))/(2*A)
Q=MASSD/2*(MASSA*VELOCA/MASSD)**2+((MASSA*VELOCA)**2)/2
```

Note that brackets have been used to make the order of evaluation more obvious. It is often possible to write involved expressions without brackets, but, for the sake of clarity, it is often best to leave the brackets in, even to the extent of inserting a few extra to ensure that the expression is evaluated correctly. The expression will be evaluated just as quickly with the brackets as without. Also note that none of the expressions are particularly complex. The last one is about as complex as you should try: with more complexity than this it is easy to make a mistake.

Problems arise when the value that a *faulty* expression yields lies within the range of expected values and the error may well go undetected. This may appear strange at first, but, a computer does exactly what it is instructed. If, through a misunderstanding on the part of a programmer, the program is syntactically correct but logically wrong from the point of view of the problem definition, then this will not be spotted by the compiler. If an expression is complex, break it down into successive statements with elements of the expression on each line, e.g.

```
TEMP = B*B-4*A*C
X1 = (–B + (TEMP**0.5))/(2*A)
```

and

```
MOMENT = MASSA * VELOCA
Q = MASSD/2*(MOMENT/MASSD)**2+(MOMENT**2)/2
```

Rounding and truncation

When arithmetic calculations are performed one of the following can occur:–

- **truncation.** This operation involves throwing away part of the number, e.g. with 14.6 truncating the number to two figures leaves 14.
- **rounding**. Consider 14.6 again. This is rounded to 15. Basically, the number is changed to the nearest whole number. It is still a real number. What do you think will happen with 14.5, will this be rounded up or down?

You must be aware of these two operations. They may occasionally cause problems in division and in expressions with more than one data type.

To see some of the problems that can occur consider the examples below:–

```
REAL A,B,C
INTEGER I
A=1.5
B=2.0
C=A/B
I=A/B
```

After executing these statements C has the value 0.75, and I has the value zero! This is an example of type conversion across the = sign. The variables on the right are all real, but the last variable on the left is integer. The value is therefore made into an integer by truncation. In this example, 0.75 is real, so I becomes zero when truncation takes place.

Consider now an example where we assign into a real variable (so that no truncation due to the assignment will take place), but where part of the expression on the right hand side involves integer division.

```
INTEGER I,J,K
REAL ANSWER
I=5
J=2
K=4
ANSWER=I/J*K
```

The value of ANSWER is 8, because the I/J term involves integer division. The expected answer of 10 is not that different from the actual one of 8, and it is cases like this that cause problems for the unwary, i.e. where the calculated result may be close to the actual one. In complicated expressions it would be easy to miss something like this.

To recap, truncation takes place in Fortran

- across an = sign, when a real is assigned to an integer;
- in integer division

It is very important to be careful when attempting *mixed mode arithmetic* – that is, when mixing reals and integers. If a real and integer are together in a division or multiplication, the result of that operation will be real; when addition or subtraction takes place, in a similar situation, the result will also be real. The problem arises when some parts of an expression are calculated using integer arithmetic, and other parts with real arithmetic:–

```
C = A + B - I / J
```

The integer division is carried out before the addition and subtraction, hence the result of I/J is integer, although all the other parts of the expression will be carried out with real arithmetic.

Example program

How long does it take for light to reach the Earth from the Sun? Light travels 9.46 10^{12} km in one year. We can take a year as being equivalent to 365.25 days. (As all school-children know, the astronomical year is 365 days, 5 hours, 48 minutes and 45.9747 seconds – hardly worth the extra effort.) The distance between the Earth and Sun is about 150,000,000 km. There is obviously a bit of imprecision involved in these figures, not least since the Earth moves in an elliptical orbit, not a circular one. One last point to note before presenting the program is that the elapsed time will be given in minutes and seconds. Few people readily grasp fractional parts of a year.

```
      PROGRAM TIME
      REAL LTYR, LTMIN, DIST, ELAPSE
      INTEGER MINUTE, SECOND
C
C LTYR  :DISTANCE TRAVELLED BY LIGHT IN ONE YEAR IN KM
C LTMIN :DISTANCE TRAVELLED BY LIGHT IN ONE MINUTE IN KM
C DIST  :DISTANCE FROM SUN TO EARTH IN KM
C ELAPSE:TIME TAKEN TO TRAVEL A DISTANCE DIST IN MINUTES
C MINUTE:INTEGER NUMBER PART OF ELAPSE
C SECOND:INTEGER NUMBER OF SECONDS EQUIVALENT TO
C        FRACTIONAL PART OF ELAPSE
C
      LTYR=9.46*10**12
      LTMIN=LTYR/(365.25*24.0*60.0)
      DIST=150.0*10**6
C
      ELAPSE=DIST/LTMIN
      MINUTE=ELAPSE
      SECOND=(ELAPSE-MINUTE)*60
C
      PRINT *,' LIGHT TAKES ',MINUTE,' MINUTES'
      PRINT *,'             ',SECOND,' SECONDS'
      PRINT *,' TO REACH THE EARTH FROM THE SUN'
      END
```

The calculation is straightforward; first we calculate the distance travelled by light in one minute, and then use this value to find out how many minutes it takes for light to travel a set distance. Separating the time taken in minutes into whole number minutes and seconds is accomplished by exploiting the way in which Fortran will truncate a real number to an integer on type conversion. The difference between these two values is the part of a minute which needs to be converted to seconds. Given the inaccuracies already inherent in the exercise, there seems little point in giving decimal parts of a second.

It is worth noting that some structure has been attempted by using comment lines to separate parts of the program into fairly distinct chunks. Note also that the comment lines describe the variables used in the program.

The PARAMETER statement.

This statement is used to provide a way associating a meaningful name with a *constant* in a program. Consider a program where PI was going to be used a lot. It would be silly to have to type in 3.14159265358 etc. every time. There would be a lot to type and it is likely that a mistake could be made typing in the correct value. It therefore makes sense to set up PI once and then refer to it by name. However, if PI was just a variable then it would be possible to do the following:–

```
REAL LI,PI
.
PI=3.14159265358
.
.
PI=4*ALPHA/BETA
.
.
.
```

The PI=4*ALPHA/BETA statement should have been LI=4*ALPHA/BETA. What has happened is that, through a typing mistake (P and L are close together on a keyboard), an error has crept into the program. It will not be spotted by the compiler. Fortran provides a way of helping here with the PARAMETER statement, which should be preceded with a type declaration. The following are correct examples of the PARAMETER statement:–

```
REAL PI,C
PARAMETER (PI=3.14159265358,C=2.997925)
```

and

```
REAL CHARGE
PARAMETER (CHARGE=1.6021917)
```

The advantage of the PARAMETER statement is that you could not then assign another value to PI, C or CHARGE. If you tried to do this, the compiler would generate an error message.

A PARAMETER statement may contain an arithmetic expression, so that some relatively simple arithmetic may be performed in setting up these constants. The evaluation must be confined to addition, subtraction, multiplication, division and integer exponentiation. The following examples help to demonstrate the possibilities

```
REAL PARSEC,PI,RADIAN
PARAMETER (PARSEC=3.08*10**16)
PARAMETER (PI=3.14159265358,RADIAN=360./PI)
```

Precision and size of numbers

The precision and the size of a number in computing is directly related to the number of bits allocated to its internal representation. On a large number of computers this is the same as the word size. The following summarises this information for three machines.

Machine and word size (bits)	Maximum Integer	Smallest real Largest real
Cray (64)	(2**63)–1 70368744177663	10.0E-2466 0.13634E+2466
CDC (60)	(2**48)-1 281474976710655	3.131513062514E-294 1.265014083171E+322
VAX (32)	(2**31)-1 2147483647	0.29E-38 1.7E38
IBM PC (8, 16 and 32)	(2**31)-1 2147483647	0.29E-38 1.7E38

Precision is not the same as accuracy. In this age of digital time-keeping, it is easy to provide an extremely precise answer to the question *What time is it*? This answer need not be accurate, even though it is reported to tenths (or even hundredths!) of a second. Do not be fooled into believing that an answer reported to ten places of decimals must be accurate to ten places of decimals. The computer can only retain a limited precision. When calculations are performed, this limitation will tend to generate inaccuracies in the result. The estimation of such inaccuracies is the domain of the branch of mathematics known as Numerical Analysis.

To give some idea of the problems, consider an imaginary *decimal* computer, which retains two significant digits in its calculations. For example, 1.2, 12.0, 120.0 and 0.12 are all given to two digit precision. Note therefore that 1234.5 would be represented as 1200.0 in this device. When any arithmetic operation is carried out, the result (including any intermediate calculations) will have two significant digits. Thus

```
130+12 = 140 (rounding down from 142)
```

and similarly

```
17/3 = 5.7 (rounding up from 5.666666...)
```

and

```
16*16 = 260
```

Where there are more involved calculations, the results can become even less attractive. Assume we wish to evaluate

```
(16*16) / 0.14
```

We would like an answer in the region of 1828.5718, or, to two significant digits, 1800.0. If we evaluate the terms within the brackets first, the answer is 260/0.14, or 1857.1428; 1900.0 on the two digit machine. Thinking that we could do better, we could re-write the fraction as

```
(16/0.14) * 16
```

This gives a result of 1800.0.

Algebra shows that all these evaluations are equivalent if unlimited precision is available.

Care should also be taken when is one is near the numerical limits of the machine. Consider the following

```
Z = B * C / D
```

where B, C and D are all 10^{30} and we are using a VAX or IBM PC where the maximum real is approximately 10^{38}. Here the product B * C generates a number of 10^{60} – beyond the limits of the machine. This is called *overflow* as the number is too large. Note that we could avoid this problem by retyping this as

```
Z = B * (C/D)
```

where the intermediate result would now be $10^{30}/10^{30}$, i.e. 1.

There is an inverse called *underflow* when the number is too small, which is illustrated below.

```
Z = X1 * Y1 * Z1
```

where X1 and Y1 are 10^{-20} and Z1 is 10^{20}. The intermediate result of X1 * Y1 is 10^{-40} – again beyond the limits of the machine. This problem could have been overcome by retyping as

```
Z = X1 * (Y1 * Z1)
```

This is a particular problem for many scientists and engineers with all machines that use 32 bit arithmetic for integer and real calculations. This is because many physical constants, etc are around the limits of the magnitude (large or small) supported by single precision. This is rarely a problem with the Cray or CDC machines.

Summary

- learn the rules for the evaluation of arithmetic expressions;
- break expressions down where necessary to ensure that the expressions are evaluated in the way you want;
- take care with truncation due to integer division in an expression. Note that this will only be a problem where both parts of the division are INTEGER;
- take care with truncation due to the assignment statement when there is an integer on the left hand sign of the statement, i.e. assigning a real into an integer variable;
- when you want to set up constants, which will remain unchanged throughout the program, use the PARAMETER statement;
- do not confuse precision and accuracy.

Problems

1. Modify the program that read in your name and address to also read in and print out your age, telephone number and sex.

2. One of the easiest ways to write a program is to modify an existing one. The example given earlier, dealing with the time taken for light to travel from the Sun to the Earth, could form the basis of several other programs.

(i) Many communications satellites follow a geosynchronous orbit, some 35,870 km above the Earths surface. What is the time lag incurred in using one such satellite for a telephone conversation?

(ii) The Moon is about 384,400 km from the Earth on average. What implications does this have for control of experiments on the Moon? What is the time lag?

(iii) The following table gives the distance in Mkm from the Sun to the planets in the Solar System:

Mercury	57.9
Venus	108.2
Earth	149.6
Mars	227.9
Jupiter	778.3
Saturn	1427
Uranus	2869.6
Neptune	4496.6
Pluto	5900

Use this information to find the greatest and least time taken to send a message from Earth to the other planets. Assume that all orbits are in the same plane and circular (if it was good enough for Copernicus, its good enough for this example). For all practical purposes, the speed of light in vacuum is a constant, and therefore an excellent candidate for a PARAMETER statement. Use it.

3. Write a program to read in, sum and average five numbers.

4. Write a program to calculate the period of a pendulum. Use the following formula:–

```
T=2*PI*(LENGTH/9.81)**.5
```

Calculate the period for at least 5 values of the length. The length (LENGTH) is in metres, and the time (T) in seconds.

5. Unit pricing: the following table gives the price and weight of various cereals available in the local supermarket.

Cereal	Price	Weight (grams)
Frostys	75	375
Special L	76	250
Rice Crispys	71	295
Rice Crispys	97	440
Bran Bits	85	625
Raisin Bran	84	375
Icicles	67	280
Frostys	58	250
Coco Puffs	77	280
Huffa Puffa Rice	76	230
Friends Oats	74	750
Weetabits	40	375
Welsh Porage Oats	74	750
More	61	250
Korn Flakes	81	500
Korn Flakes	65	375

Which of these gives best value for money, in terms of cost per gram (i.e. unit pricing)? Which gives the poorest value, on the same criterion?

5. Write a program that tests the precision and size of numbers of the system you use. Finding out the word size of the machine you work with is the first step. Then try some multiplication and division, and see what sort of error messages you get as the numbers become too small and too large.

7

Arrays and DO loops

Thy gifts, thy tables, are within my brain
Full charactered with lasting memory

William Shakespeare, 'The Sonnets'

Aims

The aims of this chapter are:–

- to introduce the ideas of tables of data and some of the formal terms used to describe them
    - Array
    - Vector
    - List and linear list
- to discuss the array as a random access structure where any element can be accessed as readily as any other
- to note that the data in an array is all of the same type
- to introduce the twin concepts of data structure and corresponding control structure
- to introduce the statements necessary in Fortran to support and manipulate these data structures

Tables of data

Consider the following:–

¤ **Telephone directory**

A telephone directory consists of the following kinds of entries:–

Name	Address	Number
Adcroft A.	61 Connaught Road, Roath, Cardiff223309	
Beale K.	14 Airedale Road, Balham	745 9870
Blunt R.U.	81 Stanlake Road, Shepherds Bush	674 4546
...		
...		
...		
Sims Tony	99 Andover Road,Twickenham	898 7330

This *structure* can be considered in a variety of ways, but perhaps the most common is to regard it as a *table* of data, where there are 3 columns and as many rows as there are entries in the telephone directory.

Consider now the way we extract information from this table. We would scan the name column looking for the name we are interested in, and then read along the row looking for either the address or telephone number, i.e. we are using the name to *look up* the item of interest.

¤ **Book catalogue**

An catalogue could contain:–

Author(s)	Title	Publisher
Carrol L.	Alice through the looking Glass	Penguin
Knuth D.	Semi-numerical Algorithms	Addison-Wesley
...		
Steinbeck.J	Sweet Thursday	Penguin
...		
Wiederhold G.	Database Design	McGraw-Hill

Again, this can be regarded as a *table* of data, having 3 columns and many rows. We would follow the same procedure as with the telephone directory to extract the information. We would use the *Name* to *look up* what books are available.

¤ **Examination marks or results**

This could consist of:–

Name	Physics	Maths	History	Geography	French
Fowler .L.	50	47	89	30	46
Barron.L.W	37	67	65	68	98
Warren.J.	25	45	48	10	36
Mallory.D.	89	56	45	30	65
Codd.S.	68	78	76	98	65

This can again be regarded as a *table* of data. This example has 6 columns and 5 rows. We would again *look up* information by using the Name.

¤ **Monthly rainfall**

Typically this would consist of:–

Month	Rainfall
January	10.4
February	11.1
March	8.3
April	7.5
May	3.2
June	4.6
July	3.2
August	4.5
September	2.1
October	3.1
November	10.1
December	11.1

In this table there are 2 columns and 12 rows. To find out what the rainfall was in July, we scan the table for July in the Month column and locate the value in the same row, i.e. the rainfall figure for July.

These are just some of the many examples of problems where the data that is being considered has a tabular structure. Most general purpose languages therefore have mechanisms for dealing with this kind of structure. Some of the special names given to these structures include:–

- Linear list
- List
- Vector
- Array

The term used most often here, and in most books on Fortran programming, is *array.*

Arrays in Fortran

There are three key things to consider here:–

- the ability to refer to a set or group of items by a single name;
- the ability to refer to individual items or members of this set, i.e. look them up;
- the choice of a control structure that allows easy manipulation of this set or array.

The DIMENSION statement

The DIMENSION statement defines a variable to be an array. This satisfies the first requirement of being able to refer to a set of items by a single name.

Some examples are given below:–

```
DIMENSION WAGES(100)
DIMENSION SAMPLE(10000), MATRIX(100,100)
DIMENSION PHASE(10,10,10,2)
```

In each of these examples, the integer values in brackets specify the maximum number of items which may be kept in the array. In the case of WAGES, up to 100; in the case of PHASE, up to 2000.

An index

An index enables you to refer to or select individual elements of the array. In the telephone directory, book catalogue, and exam marks table we used the name to index or look up the items of interest; in the monthly rainfall example we used the month name to index or look up the item of interest.

Control structure

The statement that is generally used to manipulate the elements of an array is the DO statement. It is typical to have several statements controlled by the DO statement, and the block of repeated statements is often called a *DO loop*.

Let us look at two complete programs that highlight the above.

Monthly Rainfall

Let us look at this earlier example in more depth now. Consider the following:–

Month	Associated integer representation	Array and index	Rainfall value
January	1	Rainfl(1)	10.4
February	2	Rainfl(2)	11.1
March	3	Rainfl(3)	8.3
April	4	Rainfl(4)	7.5
May	5	Rainfl(5)	3.2
June	6	Rainfl(6)	4.6
July	7	Rainfl(7)	3.2
August	8	Rainfl(8)	4.5
September	9	Rainfl(9)	2.1
October	10	Rainfl(10)	3.1
November	11	Rainfl(11)	10.1
December	12	Rainfl(12)	11.1

Most of you should be familiar with the idea of the use of an integer as an alternate way of representing a month, e.g. in a date expressed as 1/3/1989, for 1^{st} March 1989 (anglicised style) or 3^{rd} January (americanised style). Fortran, in common with other older programming languages, only allows the use of integers as an index into an array. Thus when we write a program to use arrays we have to map between whatever construct we use in everyday life as our index (names in our examples of telephone directory, book catalogue, and exam marks) to an integer representation in Fortran.

The following program reads in the 12 monthly values from the terminal, computes the sum and average for the year, and prints the average out.

```
      PROGRAM RAIN
      REAL RAINFL, SUM, AVERGE
      DIMENSION RAINFL(12)
      INTEGER MONTH
      PRINT *,' Type in the rainfall values'
      PRINT *,' one per line'
      DO 10 MONTH=1,12
         READ *, RAINFL(MONTH)
10    CONTINUE
      ...
      ...
      DO 20 MONTH=1,12
         SUM = SUM + RAINFL(MONTH)
20    CONTINUE
      AVERGE = SUM / 12
      PRINT *,' Average monthly rainfall was'
      PRINT *, AVERGE
      END
```

RAINFL is the *array name.* The variable MONTH in brackets is the index. It takes on values from 1 to 12 inclusive, and is used to pick out or select elements of the array. The index is thus a variable and this permits dynamic manipulation of the array at *run time.*

The general form of the DO statement is:

```
      DO label Counter = Start, End, Increment
```

The block of statements that form the loop are contained between the DO statement, which marks the beginning of the block or loop, and the CONTINUE statement with its associated label, which marks the end of the block or loop. In Fortran, labels are integer items which occur in columns 1 to 5. They may be placed anywhere in these columns, but may not stray into column 6. Any blanks in labels (and in statements too) are ignored; the following are therefore equivalent:–

```
197         CONTINUE
1 97  CONTINUE
197   CONTINUE
197   C O N T I N U E
```

Any character other than a blank or digit in a label will cause an error. Note that the labels 010 and 10 are considered equivalent — the *leading* zero is ignored.

In this program, the DO loops took the form:–

```
      DO 10 MONTH=1,12        start
                                   body
10    CONTINUE                     end
```

and

```
        DO 20 MONTH=1,12              start
        ...
        ...                                 body
        ...
20      CONTINUE                            end
```

The body of the loop in the program above has been indented. This is *not* required by Fortran. However it is good practice and will make programs easier to follow.

The number of times that the DO loop is executed is governed by the last part of the DO statement, i.e. by the:–

```
        Counter = Start, End, Increment
```

Start, as it implies, is the initial value which the counter (or index, or control variable) takes. Each time the loop is executed, the value of the counter will be increased by the value of *increment,* until the value of *end* is reached.

If *increment* is omitted, it is assumed to be 1. No other element of the DO statement may be omitted. In order to execute the statements within the loop (the *body*) it must be possible to reach *end* from *start.* Thus zero is an illegal value of *increment.* In the event that it is not possible to reach *end,* the loop will not be executed and control will pass to the statement after the end of the loop.

In the examples above both loops would be executed 12 times. In both cases, the first time round the loop the variable MONTH would have the value 1, the second time round the loop the variable MONTH would have the value 2 etc., and the last time round the loop MONTH would have the value 12. It is customary to restrict the DO loop *counter, start, end* and *increment* variables to integer values.

Peoples Weight's

Consider the following:–

Person	Associated integer representation	Array and index	Associated value
Andy	1	WEIGHT(1)	48.7
Barry	2	WEIGHT(2)	76.5
Cathy	3	WEIGHT(3)	58.5
Dawn	4	WEIGHT(4)	65.3

Elaine	5	WEIGHT(5)	88.7
Frank	6	WEIGHT(6)	67.5
Gordon	7	WEIGHT(7)	56.7
Hannah	8	WEIGHT(8)	66.7
Ian	9	WEIGHT(9)	70.6
Jatinda	10	WEIGHT(10)	99.9

We have ten people, with their names as shown. We associate each name with a number — in this case we have ordered the names alphabetically, and the numbers therefore reflect their ordering. WEIGHT is the *array name*. The number in brackets is called the *index*. The index is used to pick out or select elements of the array. Therefore the table is read as ‘the first element of the array WEIGHT has the value 48.7, the second element of the array WEIGHT has the value 76.5.

There are two examples in the program below:-

```
      PROGRAM SUMAVE
C
C THE PROGRAM READS UP TO 10 WEIGHTS INTO THE
C ARRAY WEIGHT
C VARIABLES USED
C    WEIGHT, HOLDS THE WEIGHT OF THE PEOPLE
C    PERSON, AN INDEX INTO THE ARRAY
C    TOTAL, TOTAL WEIGHT
C    AVERAG, AVERAGE OF THE WEIGHTS
C
C THE WEIGHTS ARE WRITTEN OUT SO THAT THEY CAN BE CHECKED
C

      REAL WEIGHT,TOTAL,AVERAG
      INTEGER PERSON
      DIMENSION WEIGHT(10)
      TOTAL=0.0
      DO 100 PERSON=1,10
         READ *,WEIGHT(PERSON)
         TOTAL = TOTAL + WEIGHT(PERSON)
100   CONTINUE
      AVERAG = TOTAL / 10
      PRINT *,' SUM OF NUMBERS IS   ',TOTAL
      PRINT *,' AVERAGE WEIGHT IS   ',AVERAG
      PRINT *,' 10 WEIGHTS WERE  '
      DO 200 PERSON=1,10
         PRINT *,WEIGHT(PERSON)
200   CONTINUE
      END
```

Higher dimension arrays

There are many instances where it is necessary to have arrays with more than one dimension. Take the following two examples:

¤ **A Map**

Consider the representation of a map as a set of numbers. This might be done as:–

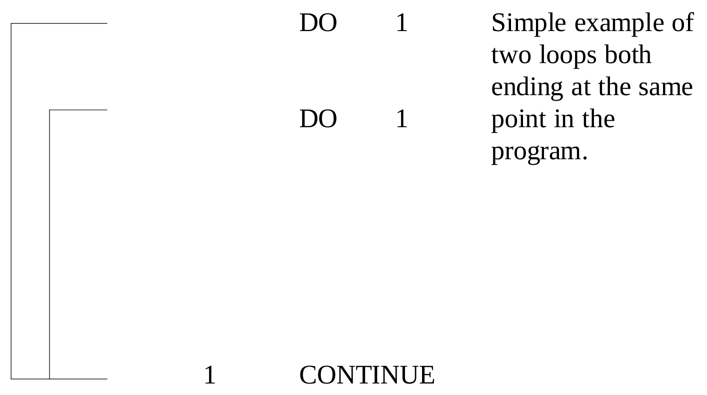
	Latitude	1	2	3	4	5
Longitude						
	1	11.113.214.515.616.7				
	2	12.113.412.111.811.7				
	3	12.013.312.811.711.4				
	4	11.813.012.611.411.3				
	5	11.312.512.311.010.9				

The values in the *array* are the heights above sea level. A program to manipulate this data structure would involve something like the following:–

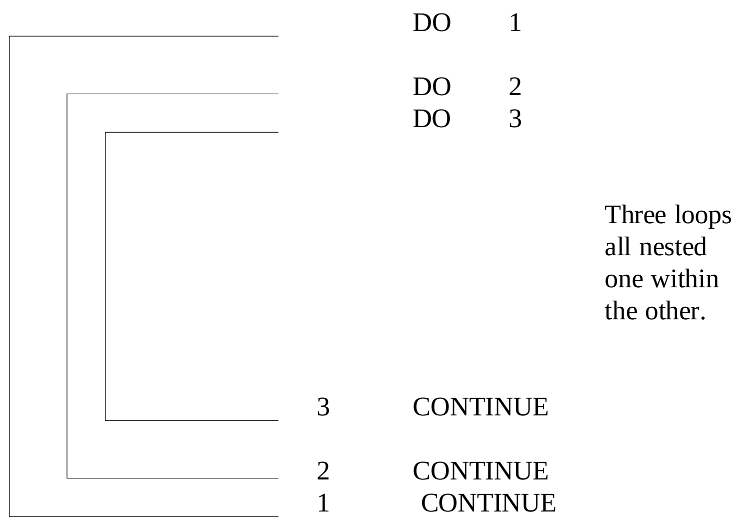
```
      PROGRAM LOC8
C
C VARIABLES USED
C HEIGHT - USED TO HOLD THE HEIGHTS ABOVE SEA LEVEL
C LONGIT - USED TO REPRESENT THE LONGITUDE
C     RESTRICTED TO INTEGER VALUES.
C     AGAIN RESTRICTED TO INTEGER VALUES.
      REAL HEIGHT
      INTEGER LONGIT,LATUDE
      DIMENSION HEIGHT(5,5)
      .
      .
      .
      DO 10 LATUDE=1,5
         DO 11 LONGIT=1,5
            PRINT *,HEIGHT(LONGIT,LATUDE)
11       CONTINUE
10    CONTINUE
      .
      .
```

Note the way in which indentation has been used to highlight the structure in this example. The *inner* loop is said to be *nested* within the outer one. It is very common to encounter problems where nesting is a natural way to express the solution. Nesting is permitted to any depth.

Here are two examples of valid nested DO loops.



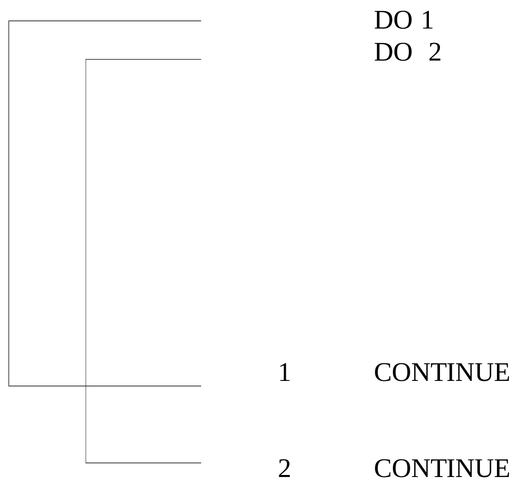
```
                DO      1       Simple example of
                                two loops both
                                ending at the same
                DO      1       point in the
                                program.



          1     CONTINUE
```



```
                DO      1

                DO      2
                DO      3

                                Three loops
                                all nested
                                one within
                                the other.

          3     CONTINUE

          2     CONTINUE
          1      CONTINUE
```

The first example shows that more than one loop may end at the same statement. In general, we do not recommend this, not least since it makes indentation rather awkward. Extra labels do not cost anything, and generally increase the comprehensibility of a program.

The example below is *illegal.* A moment's thought shows that this must be the case; the effect of repeating statements has no meaning when the loops cross one another.



```
                DO 1
                DO  2




        1       CONTINUE

        2       CONTINUE
```

Booking arrangements in a theatre or cinema

A theatre or cinema consists of rows and columns of seats. In a large cinema or a typical theatre there would also be more than one level or storey. Thus, a program to represent and manipulate this structure would probably have a two or three dimensional array.

Consider the following program extract:–

```
      PROGRAM THEATR
      INTEGER ROW,COLUMN,FLOOR,NROWS,NCOLS,NFLOOR
      DIMENSION SEATS(30,30,3)
      .
      .
      .
      DO 100 FLOOR=1,NFLOOR
         DO 101 COLUMN=1,NCOLS
            DO 102 ROW=1,NROWS
            PRINT *,SEATS(ROW,COLUMN,FLOOR)
102         CONTINUE
101      CONTINUE
100   CONTINUE
      ...
      ...
      ...
```

An interesting question here is what is the best data type for *SEATS*. We will leave the choice to you.

Summary

The DIMENSION statement declares a variable to be an array. The DIMENSION statement must come at the start of a program unit, with other *declarative* statements. To recap the statements covered so far, the order is summarised below.

PROGRAM	first statement
INTEGER REAL CHARACTER DIMENSION	in any order *declarative*
PARAMETER	after other declarative
Arithmetic assignment DO CONTINUE	in any order *executable*
END	last statement

- The statement used most often to manipulate arrays is the DO statement.
- Arrays can have up to 7 dimensions
- DO loops may be nested, but they must not overlap.

Problems

1. Using a DO loop and an array rewrite the program which calculated the average of five numbers (question 3 in chapter 6) and increase the number of values read in from 5 to 10.

2. Modify this program to sum and average people's weights.

3. Generalise this program by allowing an arbitrary number of weights to be read in. What is a sensible upper bound here?

4. Modify the program that read in your name to read in 10 names. Use an array and a DO loop. When you have read the names into the array write them out in reverse order on separate lines.

5. Combine the programs that read in and calculate the average weight with the one that reads in peoples names. The program should read in the weights into one array, and the names into another. Allow 20 characters for the length of a name. Print out a table linking names and weights, i.e. something like

```
Person          Weight
_____________________

Andy            48.7
Barry           76.5

...

...
```

6. How many distinct labels could you have in a program?

8

Arrays and DO loops (2)

Here, take this book, and peruse it well:
The iterating of these lines brings gold;

Christopher Marlowe, ‘The Tragical History of Doctor Faustus’

Aims

The aims of this chapter are:–

- to extend the ideas covered in the first chapter on arrays, with the aid of several concrete examples
- to introduce an extended form of the DIMENSION statement
- to introduce the corresponding alternative form to the DO statement, to help manipulate the array in this new form
- to introduce the DO loop as a mechanism for the control of repetition in general, not just for manipulating arrays

Example 1

Consider the problem of an experiment where the independent variable voltage varies from –20 to +20 volts and the current is measured at 1 volt intervals. Fortran has a mechanism for handling this type of problem:–

```
        PROGRAM RESULT
        DIMENSION CURRNT (–20:20)
        REAL CURRNT,RESIST
        INTEGER VOLTAG
        .
        .
        .
        DO 10 VOLTAG=–20,20
            .
            CURRNT(VOLTAG)=VOLTAG/RESIST
            .
            .
10      CONTINUE
        .
        .
        END
```

We appreciate that, due to experimental error, the voltage will not have exact integer values. However, we are interested in representing and manipulating a set of values, and thus from the point of view of the problem solution and the program this is a reasonable assumption.

There are several things to note here:–

• This form of the DIMENSION statement–

```
        DIMENSION CURRNT(FIRST:LAST)
```

is of considerable use when the problem has an effective index which does not start at 1 (as implied by the original form of the statement).

• There is a corresponding form of the DO statement which allows processing of problems of this nature. This is shown in the above program. The general form of the DO statement is therefore:–

```
        DO label counter=start, end, increment
```

where *start*, *end* and *increment* can be positive or negative.

Note that zero is a legitimate value of the dimension limits and of a DO loop index.

Example 2

Consider the problem of the production of a table linking time difference with longitude. The values of longitude will vary from –180 to +180 degrees, and the time will vary from +12 hours to –12 hours. A possible program segment is:–

```
          PROGRAM ZONE
          DIMENSION TIME(–170:180)
          REAL TIME
          INTEGER DEGREE,STRIP
          .
          .
          DO 10 DEGREE=–170,180,10
               VALUE=DEGREE/15.
               DO 11 STRIP=0,9
                    TIME(DEGREE+STRIP)=VALUE
11              CONTINUE
10        CONTINUE
          .
          .
          END
```

Notes

- The values of the time are **not** being calculated at every degree interval.
- The variable TIME is a real variable. It would be possible to arrange for the time to be an integer by expressing it in either minutes or seconds.
- This example takes no account of all the wiggly bits separating time zones, or of British Summer Time.

Example 3

Consider the production of a table of temperatures. The independent variable is the Fahrenheit value; the Celsius temperature is the dependent variable. Strictly speaking, a program to do this does not have to have an array, i.e. the DO loop can be used to control the repetition of a set of statements that make no reference to an array.

The following page shows a possible program segment.

```
      PROGRAM CONVRT
      INTEGER FAHREN
      REAL CELSIU
      .
      .
      DO 100 FAHREN=-100,200
        .
        CELSIU=(FAHREN-32)*5./9.
        PRINT *,FAHREN,CELSIU
        .
100   CONTINUE
      .
      END
```

Note here that the DO statement has been used *only* to control the repetition of a block of statements. In the conversion of Fahrenheit to Celsius, we multiply by 5./9. rather than 5/9; do you recall why?

This is the other use of the DO statement. The DO loop thus has two functions, its use with arrays as a control structure, and its use solely for the repetition of a block of statements.

Example 4

In the calculation of the mean and standard deviation of a list of numbers, we may use the following formulae It is not actually necessary to store the values, nor to accumulate the sum of the values and their squares. In the first case, we would possibly require a large array, while in the second, it is conceivable that the accumulated values (especially of the squares) might be too large for the machine. The following example uses an *updating* technique which avoids these problems, but is still accurate. The DO loop is simply a control structure to ensure that all the values are read in, with the index being used in the calculation of the updates.

```
      PROGRAM MEANSD
C
C VARIABLES USED ARE
C    MEAN - FOR THE RUNNING MEAN
C    SSQ  - THE RUNNING CORRECTED SUM OF SQUARES
C    X    - INPUT VALUES FOR WHICH MEAN AND SD REQUIRED
C    W    - LOCAL WORK VARIABLE
C    SD   - STANDARD DEVIATION
C    R    - ANOTHER LOCAL WORK VARIABLE
C
      REAL MEAN,SSQ,X,W,SD,R
      INTEGER I
      MEAN=0.0
      SSQ=0.0
      PRINT *,' ENTER THE NUMBER OF READINGS'
      READ*,N
      PRINT*,' ENTER THE ',N,' VALUES, ONE PER LINE'
      DO 1 I=1,N
         READ*,X
         W=X-MEAN
         R=I-1
         MEAN=(R*MEAN+X)/I
         SSQ=SSQ+W*W*R/I
1     CONTINUE
      SD=(SSQ/R)**0.5
      PRINT *,' MEAN IS ',MEAN
      PRINT *,' STANDARD DEVIATION IS ',SD
      END
```

Summary

• The DIMENSION statement allows limits to be specified for a block of information which is to be treated in a common way. The limits must be integer, and the second limit must exceed the first e.g.

```
      DIMENSION LIST(–123:–10)
      DIMENSION X(0:2000)
      DIMENSION VALUE(1:100)
```

The last example could equally be written

```
      DIMENSION VALUE(100)
```

where the first limit is omitted, and is given the default value 1. The array LIST would contain 114 values, while X would contain 2001.

• A DO statement and its corresponding CONTINUE statement define a loop. The DO statement provides a starting value, terminal value, and, optionally, an increment, for its index or counter.

• Although these values need not be integers, you are strongly advised to make them so. The increment may be negative, but should never be zero. If it is not present, the default value is 1. It must be possible for the terminating value to be reached from the starting value.

• The counter in a DO loop is ideally suited for indexing an array, but it may be used anywhere that repetition is needed, and of course the index or counter need not be used explicitly.

Problems

1. Write a program to print out a table of values for the conversion from litres to pints.

2. Write a program to print out the 12 times table. Typical output would be of the form:

```
         1       *      12      =      12
         2       *      12      =      24
         3       *      12      =      36
```

etc.

3. Complete the program which calculates a table of Fahrenheit and corresponding Celsius temperatures. Employ the PARAMETER statement for the constant term.

4. In order to obtain some impression of the inaccuracies generated by limited precision, try the following. Take the square root of an input value a given number of times (i.e. raise it to the power 0.5); recreate the the original value by squaring the resulting value the same number of times. Intuitively, the starting and recreated value should be the same, but the limited machine precision will lead to some inaccuracy.

9

Output: An Introduction

Why, sometimes I've believed as many as six impossible things before breakfast.

Lewis Carroll, 'Alice through the Looking-Glass'

Aims

The aims here are to introduce the facilities for producing neat output, and to show how to write results to a file, rather than to the terminal. In particular

- the A, I, E, F, and X layout or edit descriptors
- the OPEN, WRITE, and CLOSE statements

Introduction

When you have used PRINT * a few times it becomes apparent that it is not always as useful as it might be. The data is written out in a way which makes some sense, but may not be especially easy to read. Real numbers are written out with all their significant places, which is very often rather too many, and it is often difficult to line up the columns for data which is notionally tabular. It is possible to be much more precise in describing the way in which information is presented by the program. To do this, we use FORMAT statements. Through the use of the FORMAT, we can specify how many columns a number should take up, and, where appropriate, where a decimal point should lie. The FORMAT statement has a label associated with it; through this label, the PRINT statement associates the data to be written with the form in which to write it. For example:–

```
        PRINT 100,I, XVALUE(I), YVALUE(I)
100     FORMAT(1X,I3,2X,F7.4,2X,F6.4)
```

The label 100 which follows the PRINT statement takes the place of the asterisk we have been using up to now, and links the PRINT with the FORMAT statement. Although the FORMAT follows the PRINT in this example, this is not obligatory. However it is recommended that these statements are always kept together because when errors occur (which they inevitably do!) it will save you time and effort locating the possible source of the error.

The next thing to consider is the FORMAT statement itself, and its contents, i.e. the bits in brackets. The ‘X’ is used for generating spaces in the output, the ‘I’ is used when you want to print out an integer, and the ‘F’ is used when you want to print out real numbers.

Integers, I format

Integer format is reasonably straightforward, and offers clues for formats used in describing other numbers. I3 is an integer taking three columns. The number is right justified, a bit of jargon meaning that it is written as far to the right as it will go, so that there are no trailing or following blanks

```
423
-22
  9
```

are all right justified integers (in I3). Note that the minus sign counts as part of the number, and takes up one of the three columns we have specified here. The only problem with right justification is that, when you try to write something which looks useful, such as

```
THERE ARE 3 IMAGES AVAILABLE FOR PROCESSING
```

the integer part must be assigned a fixed size. In the above example, we might have given the field I2 to the number of images. This is fine when there are 0 to 9 images, but when there are more than 9, the message might read:–

```
THERE ARE12 IMAGES AVAILABLE FOR PROCESSING
```

which does not look very tidy, and is more difficult to read. Another alternative is to give that a very large field, say I10. But this would look extremely disjointed:–

```
THERE ARE          3 IMAGES AVAILABLE FOR PROCESSING
```

The definition of an integer format is therefore the letter I, followed by a positive integer number, e.g.

```
I1
I10
I3
I6
I12
```

Reals, F format

The F format can be seen as an extension of the integer format. But here we have to deal with the decimal point. The form of the F format specifies where the decimal point will occur, and how many digits follow it. Thus, F7.4 means that there are 4 digits after the point, in a total field width of 7 digits. Since the decimal point is also written out, there may be up to 2 digits before the decimal point. As in the case of the integer, any minus sign is part of the number, and would take up one column. Thus, the format F7.4 may be used for numbers in the range

-9.9999 to 99.9999

Let us look at the last example more closely. When a number is written out, it is rounded; that is to say, if we write out 99.99999 in an F7.4 format, the program will try to write out 100.0000! This is bad news, since we have not left enough room for all those digits before the decimal point. What happens? Asterisks will be printed. In the example above, a number out of range of the format's capabilities would be printed as:–

```
*******
```

What would a format of F7.0 do? Again, seven columns have been set aside to accommodate the number and its decimal point, but this time no digits follow the point:–

```
    99.
-21375.
```

are examples of numbers written in this format. With an F format, there is no way of getting rid of the decimal point.

The numbers making up the parts of the descriptors must all be positive integers. The definition of a real format is therefore; F followed by two integer numbers, separated by a decimal point. The first integer must exceed the second, and the second must be greater than or equal to zero. The following are valid examples:–

```
F4.0
F6.2
F12.2
F16.8
```

but these are *not* valid:–

```
F4.4
F6.8
F-3.0
F6
F.2
```

Reals, E format

The exponential or scientific notation is useful in cases where we need to provide a format which may encompass a wide range of values. If likely results lie in a very wide range, we can ensure that the most significant part is given. It is possible to give a very large F format, but alternatively, the E format may be used. This takes a form such as

```
E10.4
```

which looks something like the F, and may be interpreted in a similar way. The 10 gives the total *width* of the number to be printed out, that is the number of columns it will take. The number after the decimal point indicates the number of positions to be written after the decimal point; since all exponent format numbers are written so that the number is between 0.1 and 0.9999..., with the exponent taking care of scale shifts, this implies that the first four significant digits are to be printed out.

Taking a concrete example, 1000 may be written as 10**3, or as 0.1 * 10**4. This gives us the two parts; 0.1 gives the significant digits (in this case only one significant digit), while the 10**4 gives the exponent, namely 4 or +4. In a form that looks more like Fortran, this would be written .1E+04 where the E+04 means 10**4.

There is a minimum *size* for an exponential format. Because of all the extra bits and pieces it requires (the decimal point, the sign of the entire number, the sign of the exponent, the magnitude of the exponent and the E), the width of the number, less the number of significant places should not be less than 6. In the example given above, E10.4 meets this requirement. When the exponent is in the range zero to 99, the E will be printed as part of the number; when the exponent is greater, the E is dropped, and its place is taken by a larger value; however, the sign of the exponent is always given, whether it is positive or negative. The sign of the whole number will usually only be given when it is negative. This means that, if the numbers are always positive, the *rule of six* given above can be modified to a *rule of five*. It is safer to allow six places over, since, if the format is insufficient, all you will get are asterisks.

The most common mistake with an E format is to make the edit descriptor too small, so that there is insufficient room for all the *padding* to be printed. Formats like E8.4 just don't work (on output anyway). The following are valid E formats on output:–

```
        E9.3
        E11.2
        E18.7
        E10.4
```

but the following would not be acceptable as output formats, for a variety of reasons

```
        E11.7
        E6.3
        E4.0
        E10
        E7.3
```

The first example in this chapter could be rewritten to use E format as:–

```
        PRINT 100,I,XVALUE(I),YVALUE(I)
100     FORMAT(1X,I3,E10.4,3X,E10.4)
```

This would be of use when we are unsure about the exact range of the numbers to be printed.

Spaces

There is a shorthand way of expressing spaces (or blanks) in your output — the X descriptor, e.g.

```
        PRINT 100, ALPHA,BETA
100     FORMAT(1X,F10.4,10X,F10.3)
```

The 10X is read rather like any of the other format elements — logically it should have been X10, to correspond to I10 or F10.4, but that would be allowing intuition to run away with you. Clearly the X3J3 committee felt it important that Fortran should have inconsistencies, just like a natural language.

There are other ways of achieving the same thing — having a large space delimited by apostrophes, or by manipulating character variables. What you use is your choice. Remember that these blanks are in addition to any generated as a result of the leading blanks on numbers (if any are present). If you wish to leave a single space, you must still precede the X by a number (in this case, 1); simply writing X is illegal. The general form is therefore; positive integer followed by X.

Alphanumeric or character format, A

This is perhaps the simplest output of all. Since you will already have declared the length of a character variable in your declarations:–

```
        CHARACTER*10 B
```

when you come to write out B, the length is known — thus you need only specify that a character string is to be output:–

```
        PRINT 100,B
100     FORMAT(A)
```

If you feel you need a little extra control, you can append an integer value to the A, like A10 (A9 or A1) and so on. If you do this, only the first 10 (9 or 1) characters are written out; the remainder are ignored. Do note that 10A1 and A10 are not the same thing. 10A1 would be used to print out the first character of ten character variables, while A10 would write out the first ten characters of a single character variable. The general form is therefore just A, but if more control is required, this may be followed by a positive integer.

Within a FORMAT statement you may also write out anything within apostrophes. These strings of characters will be written out with no modification; e.g.

```
        PRINT 100,A
100     FORMAT(' THE ANSWER IS',F10.4)
```

will be written out as something like:–

```
THE ANSWER IS   12.3457
```

A partial program segment to output a 3 column table with an informative heading could be:–

```
        PRINT *,' NUMBER : X READING : Y READING'
        DO 5 I=1,100
            PRINT 100,I,XVALUE(I),YVALUE(I)
100         FORMAT(1X,I3,3X,E10.4,3X,E10.4)
5       CONTINUE
```

Common mistakes

It must be stressed that an integer can only be printed out with an I format, and a real with an F (or E) format. You cannot mix integer and F, or real and I. If you do, unpredictable results will follow.

There are (at least) two other sorts of errors you might make on writing out a value. You may try to write out something which has never actually been assigned a value; this is termed an indefinite value. You may find that the letter I is written out. In passing, note that many loaders and link editors will preset all values to zero — i.e. unset (indefinite) values are actually set to zero. On better systems there is generally some way of turning this facility off, so that undefined is really indefinite. More often than not, indefinite values are the result of mis-typing, rather than never setting values. It is not uncommon to type O for 0, or 1 for either I or l. The other likely error is to try to print out a value greater than that which the machine can calculate — *out of range* values. Some machines will print out such values as R; some machines will actually print out something which looks right, and such *overflow* and *underflow* conditions can go unnoticed. Be wary.

OPEN (and CLOSE)

One of the particularly powerful features of Fortran is the way it allows you to manipulate files. Up to now, most of the discussion has centred on reading from and writing to the terminal. It is possible to read and write to one or more files. This is achieved using the OPEN, WRITE, READ and CLOSE statements. We will consider *reading* from files in a later chapter, and concentrate on *writing* in this chapter.

The OPEN statement

This statement sets up a file for either reading or writing. A typical form is:–

```
OPEN (UNIT=1,FILE='DATA')
```

The file will be known to the operating system as DATA (or will have DATA as the first part of its name), and can be written to by using the UNIT number. This statement should come *before* you first read from or write to the file DATA.

It is not possible to write to the file DATA directly; it must be referenced through its unit number. Within the Fortran program you write to this file using a statement like

```
WRITE(UNIT=1,FMT=100) XVAL,YVAL
```

or

```
WRITE(1,100) XVAL,YVAL
```

These two statements are equivalent. Besides opening a file, we really ought to CLOSE it when we have finished writing to it:

```
        CLOSE(UNIT=1)
```

In fact, on many systems it is not obligatory to OPEN and CLOSE all your files. Almost certainly, the terminal will not require this, since INPUT and OUTPUT units will be there by default. At the end of the job, the system will CLOSE all your files. Nevertheless, explicit OPEN and CLOSE cannot hurt, and the added clarity generally assists in understanding the program.

The following program segment contains all of the above statements.

```
        PROGRAM REDO
        ...
        OPEN (UNIT=1,FILE='DATA')
        ...
        ...
        DO 100 I=1,100
            READ (UNIT=1,FMT=200) X(I)
200         FORMAT(E10.3)
            SUM = SUM + X(I)
100     CONTINUE
        ...
        CLOSE(1)
        ...
        ...
        END
```

Writing

PRINT is always directed to the file OUTPUT; in the case of interactive working, this is the terminal. This is not a very flexible arrangement. WRITE allows us to direct output to any file, including *OUTPUT*. The basic form of the WRITE is

```
        WRITE(6,100) X,Y,Z
```

or

```
        WRITE(UNIT=6,FMT=100) X,Y,Z
```

The latter form is more explicit, but the former is probably the one most widely used. We have an example here of the use of positionally dependent parameters in the first case and equated keywords in the second. With the exceptions of the PRINT statement and the READ * form of the READ, all of the input/output statements allow the unit number and the format labels to be specified either by an equated keyword (or specifier), or in a positionally dependent form. If you use the explicit UNIT= and FMT= it does not matter what order the elements are placed in, but if you omit these keywords, the unit number must come first, followed by the format label. A list of all the possible keywords is given in Chapter 18.

UNIT=6 means that the output will be written to the file given the unit number 6. In the next chapter we will cover the way in which you may associate file names and unit numbers, but, for the moment, we will assume that the default is being used. The name of the file, as defined by the system, will depend on the particular system you use; a likely name is something like DATA06, TAPE6, or FILE0006. One *easy* way to find out (apart from asking someone), is to create such a file from a program, and then look at the names of your files after the program has finished. A great many of computing's minor complexities can be clarified by simple experimentation.

FMT=100 simply gives the label of the format to be used.

The overworked asterisk may be used, either for the unit, or for the format:–

UNIT=* will write to OUTPUT (the terminal), and

FMT=* will produce output controlled by the list of variables, often called *list directed output*.

The following three statements are therefore equivalent:–

```
WRITE(UNIT=*,FMT=*) X,Y,Z
WRITE(*,*) X,Y,Z
PRINT*,X,Y,Z
```

There are other controls possible on the WRITE, which will be elaborated later.

Summary

• You have been introduced in this chapter to the use of format or layout descriptors which will allow greater control over output.

• The main features are the I format for integer variables, the E and F formats for real numbers, and the A format for characters. In addition the X, which allows insertion of spaces, has been introduced.

• Output can be directed to files as well as to the terminal, through the WRITE statement.

• The WRITE, together with the OPEN and CLOSE statements, also introduces the class of Fortran statements which use equated keywords, as well as positionally dependent parameters.

Problems

1. Write a program to produce the following kind of conversion table:–

```
CELSIUS          TEMPERATURE          FAHRENHEIT
–73.3            –100                 –148.0
```

etc.

The centre column of the table should start at –100 and go up to +100. Use F format to print out the values of CELSIUS and FAHRENHEIT, whilst the central column should use I format.

Fahrenheit temperature = (Celsius/5) * 9 +32
Celsius temperature = 5 * (Fahrenheit–32)/9

2. Write a litres and pints conversion program to produce a similar kind of output to the above. Start at 0, and make the central column go up to 50. One pint is 0.568 litres.

3. Information on car fuel consumption is usually given in miles per gallon in Britain and the US, and in litres per 100 km in Europe. Just to add an extra problem US gallons are 0.8 Imperial gallons. Prepare a table which allows conversion from either US or Imperial fuel consumption figures to the metric equivalent. Use the PARAMETER statement where appropriate.

1 Imperial gallon = 4.54596 litres
1 mile = 1.60934 kilometres.

4. Modify any of the above to write to a file rather than the terminal. What changes are required to produce a general output which will be suitable for both the terminal and a line printer? Is this degree of generality worthwhile?

5. To demonstrate your familiarity with formats, re-format questions 1, 2 or 3 to use E formats, rather than F (or vice versa).

10

Output: An Extension

Beyond the last visible dog

Russell Hoban, 'The Mouse and His Child'

Aims

The aims of this chapter are to extend the ideas introduced concerning the production of neat output, and to provide an introduction to the power and capability of the layout or edit descriptors. In particular:–

- repeated output, and implied DO loops
- formatting the output for a line-printer

Repetition

Often we need to print more than one number on a line and want to use the same layout descriptor. Consider the following:–

```
        PRINT 100,A,B,C,D
```

If each number can be written with the same layout descriptor, we can abbreviate the FORMAT statement to take account of the pattern:–

```
100     FORMAT(1X,4F8.2)
```

is equivalent to:–

```
100     FORMAT(1X,F8.2,F8.2,F8.2,F8.2)
```

as you might anticipate. If the pattern is more complex, we can extend this approach:–

```
        PRINT 100,I,A,J,B,K,C
100     FORMAT(1X,3(I3,F8.2))
```

Bracketing the description ensures that we repeat the whole entity:–

```
100     FORMAT(1X,3(I3,F8.2))
```

is equivalent to:–

```
100     FORMAT(1X,I3,F8.2, I3,F8.2, I3,F8.2)
```

Repetition with brackets can be rather more complex. In order to give some overview of formatted Fortran output, it is helpful to delve a little into the history of the language. Many of the attributes of Fortran can be traced back to the days of single user mainframes (with often a fraction of the power of many contemporary micro-computers and work-stations). These would generally take input from punched cards (the traditional 80-column Hollerith card), and would generate output on a line printer. In this sort of environment, the individual punched card had a significance which lines in a file do not have today. Each card could be seen as a single entity — a physical record unit. The *record* was seen as an element of subdivision within a file. Even then, there was some confusion between the notion of physical records and files split into logically distinct sub-units, since these sub-units might also be termed records. The present Fortran standard merely says that a record *does not necessarily correspond to a physical entity*, although *a punched card is usually considered to be a record.* This leaves us sitting at our terminals in a bemused state, especially since we may have no idea what a punched card looks like (an ideal state of affairs!)

It is important to have some notion of a record, since most of the formal definitions dealing with output (and input) are couched in terms of records. Every time an input or output statement is executed your nominal position in the file changes. If we think in terms of individual records (which may be cards), the notions of *current*, *preceding* and *next* record seem fairly straightforward. The current record is simply the one we have just read or written, and the other definitions follow naturally.

The situation becomes less clear when we realise that a single output statement may generate many lines of output.

```
        WRITE(UNIT=6,FMT=101)  A,B,C
101     FORMAT(1X,F10.4)
```

writes out three separate lines. Looking at the output alone, there is no way to distinguish this from the output generated by:–

```
        WRITE(UNIT=6,FMT=101)  A
        WRITE(UNIT=6,FMT=101)  B
        WRITE(UNIT=6,FMT=101)  C
101     FORMAT(1X,F10.4)
```

In the latter case we would probably be happy to consider each line a *record*, although in the previous example we might swither between considering all three lines (generated by a single statement) a single record or three records. Consider the first of these two examples more closely; each time the format is exhausted — that is to say, each time we run out of format description, we start again on a new line (a new record). A new record is begun as each F10.4 is begun. The correct interpretation is therefore that three records have been written.

The same sort of thing happens in more complex FORMAT statements:–

```
        WRITE(UNIT=6,FMT=105) X,I,Y
105     FORMAT(1X,F8.4,I3,(F8.4))
```

would write out a single record containing a real, an integer and a real. Using the same format statement with WRITE (UNIT=6, FMT=105) X,I,Y,Z would write out two records. The first containing the values of X, I and Y, the second containing only Z. If there were still more values

```
        WRITE(UNIT=6,FMT=105) X,I,Y,Z,A
```

would print out three records. The group in brackets — the (F8.4) — is repeated until we run out of items.

Some more examples

Since it is the last open bracket which determines the position at which the format is repeated, simply writing:–

```
        WRITE(UNIT=6,FMT=100)  A,I,B,C,J
100     FORMAT(1X,F8.4,I3,F8.2)
```

would imply that A, I and B would be written on one line, then, returning to the last open brackets, (in this case the only open brackets), a new record (or line) is begun to write out C and J. A statement like:–

```
100     FORMAT(1X,(F8.4),I3,F8.2)
```

would return to the (F8.4) group, and then continue to the I3 and F8.2 before repeating again (if necessary). The same thing happens if the (F8.4) had no brackets around it. On the other hand:–

```
100     FORMAT(1X,(F8.4),I3,(F8.2))
```

contains superfluous brackets around the F8.4, since the repeat statement will never return to that group. Are you confused yet? This seems all very esoteric, and really, we have only hinted at the complexity which is possible. It is seldom that you have to create complex FORMAT statements, and clarity is far more important than brevity.

When patterned or repeated output is used, we may want to stop when there are no more numbers to write out. Take the following example:–

```
        WRITE(UNIT=1,FMT=100) A,B,C,D
100     FORMAT(1X,4(F6.1,’,’))
```

This will give output which looks like:–

```
 37.4,  29.4,  14.2,  –9.1,
```

The last comma should not be there. We can suppress these unwanted elements by using the colon:–

```
100     FORMAT(1X,4(F6.1:’,’))
```

which would then give us:–

```
 37.4,  29.4,  14.2,  -9.1
```

Since we run out of data at the fourth item, D, the output following is not written out. It is a small point, but it does look a lot tidier. There are other ways of achieving the same thing.

This helps to illustrate another point, namely that you may have formats which are more extensive than the lists which reference them:–

```
        WRITE(UNIT=1,FMT=100) A,B,C
        WRITE(UNIT=1,FMT=100) X,Y
100     FORMAT(1X,6F8.2)
```

Both WRITE statements use the format provided, although they write out different amounts of data, and neither uses up the whole format.

Implied DO loops

In reading and writing it is possible to use more compact ways of indicating that an array is being referenced, since it is often rather tedious to declare each element which is involved, e.g.

```
        WRITE(UNIT=6,FMT=100)Y(1),Y(2),Y(3),Y(4)
100     FORMAT(1X,F8.4)
```

Clearly we could improve this slightly by making it into a loop:–

```
        DO 1 I=1,4
            WRITE(UNIT=6,FMT=100) Y(I)
100         FORMAT(1X,F8.4)
1       CONTINUE
```

and equally we can simplify this to:–

```
        WRITE(UNIT=6,FMT=100)(Y(I),I=1,4)
100     FORMAT(1X,F8.4)
```

where the DO loop is subsumed into the expression (the syntax is just the same as for the counter part of a DO loop, with the same rules for starting, ending and incrementing). An alternative, and yet more compact form is:–

```
        DIMENSION Y(4)
        ...
        ...
        WRITE(UNIT=6,FMT=100)Y
100     FORMAT(1X,F8.4)
```

In all these cases, the output would be the same — four numbers printed out on separate lines. The FORMAT statement is controlling this layout. Changing the format to:–

```
100 FORMAT(1X,4F8.4)
```

would change the layout in three out of the four cases outlined. Which three? Even two (or more) dimensional arrays can be written out or read in by implied loops.

```
        DIMENSION Y(10,10)
        NROWS=6
        NCOLS=7
        DO 1 J=1,NROWS
            WRITE(UNIT=6,FMT=100)(Y(I,J),I=1,NCOLS)
1       CONTINUE
```

```
100     FORMAT(1X,10F10.4)
```

may be written:–

```
        WRITE(UNIT=6,FMT=100)((Y(I,J),I=1,NCOLS),J=1,NROWS)
```

or even as:–

```
        WRITE(UNIT=6,FMT=100) Y
```

There are two points to note with this last example. Firstly, the entire contents of the array will be written; there is no scope for fine control. Secondly, the order in which the array elements are written may be a surprise. The order is that of the first subscript varying 1 to 10 (the array bound), with the second subscript as 1, then 1 to 10 with the second subscript as 2 and so on; the sequence is

```
Y(1,1)          Y(2,1)      Y(3,1)      Y(10,1)
Y(1,2)          Y(2,2)      Y(3,2)      Y(10,2)
        .
        .
Y(1,10) Y(2,10)         Y(10,10)
```

Another feature to note is that we can generate values from within a WRITE statement:–

```
        WRITE(UNIT=6,FMT=101)(I,I=0,9)
101     FORMAT(1X,10I3)
```

would produce a line like:–

```
 0  1  2  3  4  5  6  7  8  9
```

Formatting for a line-printer

There is one extension to format specifications which is relevant to line-printers. Fortran defines four special characters which have a particular effect on standard line-printers. They have an effect when they occur in the first character position of a line. This means that a line-printer which is not under your immediate control can be used to produce neat output, by sending a file to be printed on it. This has a variety of names including *spooling*, *queueing* and *routing* depending on the system. You should check with your local system for the exact mechanism to achieve this.

The special characters are +, 0, 1 and blank. To be used, they must be the first character of the output in each line — as if they were to be printed in column 1. In fact, a standard line printer never prints a character that occurs in column 1 at all.

Whenever a WRITE statement is begun, the printer *advances* to a new record; i.e. a new line is begun before any data is transferred. If the first character is a *special character,* then this will be interpreted by the line-printer. If the first character to be printed is a blank, the printer continues printing on that line. The first character is also known as the *carriage control character.*

The blank is a *do nothing special* control. It signifies that the line is to be printed as it is.

The zero indicates that you wish to leave an extra line; this is often useful in spacing out results to make the output more readable.

The 1 makes the output skip down to the top of the next page. This is clearly useful for separating logically distinct chunks of output. If you obtain a line printer listing of your compiled program, each segment will start at the top of a new page.

The plus is a *no advance* or *overprint* character. It suppresses the effect of the line advance which a WRITE generates. No new line is begun and the previous line is over-printed with the new. Overprinting can be useful especially when you wish to print out grey scale maps but its use is rather restricted. In particular, it can be a dangerous control character. If you have a format starting with a plus in a loop, you can make the printer overprint again and again and again .... and again and again, until it has hammered itself into a pulp. This is not a good idea.

Similarly, accidental use of the 1 as a control character in a loop will give you lots of blank pages. It is just a bit embarrassing to be presented with a six inch stack of paper which is (almost) blank, because you had a 1 repeatedly in column 1.

Mechanics of carriage control

The following are all quite reasonable ways of generating the blank in column 1:–

```
       WRITE(UNIT=6,FMT=100)A
100    FORMAT(' ',F10.4)
```

or

```
       WRITE(UNIT=6,FMT=100)A
100    FORMAT(1X,F10.4)
```

or

```
       WRITE(UNIT=6,FMT=100)A
100    FORMAT(' THE ANSWER IS ',F10.4)
```

Note, however, that

```
        WRITE(UNIT=6,FMT=100)A
100     FORMAT(F8.4)
```

could result in problems. If A contained the value 100.2934, the result on a line printer would be

```
00.2934
```

printed at the top of a new page. The 1 is taken as carriage control, and the rest of the line then printed.

Accidentally printing zeros in column 1 is a little more difficult, but:–

```
        WRITE(UNIT=6,FMT=100)I
100     FORMAT(I1)
```

might just do it. Don’t.

Remember that this only applies to line printer output, and not to the terminal. Since Fortran only defines 4 characters as carriage control, you will find that anything else in column 1 will give unpredictable results. On some systems, a fair number of alternatives may be defined by the installation, and they may do something useful. On other systems, they may do something, but they may also fail to print the rest of the line. This can be very perplexing. Beware.

Generating a new line, on both line-printers and terminals.

There are several ways of generating new lines, other than with a 0 in column one of your line printer output. A more general approach, which works on terminals and also line printers, is through the oblique or slash, /. Each time this is encountered in a FORMAT statement, a new line is begun.

```
        PRINT 101,A,B
101     FORMAT(1X,F10.4/1X,F10.4)
```

would give output like:–

```
 100.2317
 –4.0021
```

This is the same as (F10.4) would have given, but clearly this opens up lots of possibilities for formatting output more tidily:–

```
        PRINT 102,NVAL,XMAX,XMIN
102     FORMAT(’ NUMBER OF VALUES READ IN WAS: ’,I10/
   1           ’ MAXIMUM VALUE IS: ’,F10.4/
   2     ’ MINIMUM VALUE IS: ’,F10.4)
```

which may be easier to read than using only one line, and is certainly more compact to write than using three separate print statements. It is not necessary to separate / by commas, although if you do nothing catastrophic will happen. In this example we have put the elements of the format on three lines, using the continuation character in column 6.

Any statement may be extended onto a subsequent line by placing a character (with the exception of a zero) in column 6. The main reason for doing this here is that it is often difficult to guess when you have typed to column 72. It is far easier to break part of the format and restart with a continuation line. Errors in formats are often very tricky to locate, and any attempt to bring a little order will help.

You may also begin a format description with a /, in order to generate an extra line, or even generate lots of lines with lots of slashes; e.g.

```
      WRITE(UNIT=6,FMT=103)A,B
103   FORMAT(//1X,F10.4,4(/),1X,F10.4)
```

will leave two lines before printing A, and then will generate 4 new lines before writing B (i.e. there will be three lines between A and B — the fourth new line will contain B). While a slash by itself, or with another slash, does not have to be separated by commas from other groups, a more complex grouping, 4(/), does have to have commas and brackets to delimit it.

Summary

• The FORMAT statement and its associated layout or edit descriptor are powerful, and allow repetition of patterns of output (both explicitly and implicitly).

• When output is to be directed to a line-printer, there are four characters defined that allow reasonable control over the layout. Care must to be taken with these characters, since it is possible to decimate forests with little effort.

Problems

1. Modify the temperature conversion program to produce output suitable for a line-printer. Use the local operating system commands to send the file to be printed.

2. Repeat for the litres and pints program.

3. What features of Fortran reveal its evolution from punched card input?

4. Try to create a real number greater than the maximum possible on your computer — write it out. Try to repeat this for an integer. You may have to exercise some ingenuity.

5. Check what a number too large for the output format will be printed as on your local system – is it all asterisks?

6. Write a program which stores litres and corresponding pints in arrays. You should now be able to control the output of the table (excluding headings — although this could be done too) in a single WRITE or PRINT statement. If you don’t like litres and pints, try some other conversion (sterling to US dollars, leagues to fathoms, Scots miles to Betelgeusian pfnings). The principle remains the same.

11

Reading in data

Winne-the-Pooh read the two notices very carefully,
first from left to right, and afterwards,
in case he had missed some of it, from right to left.

A A Milne, Winne-the-Pooh

Aims

The aims of this chapter are to introduce some of the ideas involved in reading data into a program. In particular, using the following:–

- reading from fixed fields
- integers, reals and characters
- blanks — nulls or zeros?
- READ — extensions
  - END=
  - ERR=
- OPEN — associating unit numbers and file names
  - CLOSE
  - REWIND
  - BACKSPACE

Fixed fields on input

All the formats described earlier are available, and again they are limited to particular types. Integers may only be input by the I format, reals with F and E, and character (alphanumeric) with A.

Integers, the I format

Integers are read in with the I edit descriptor. While, on output, integers appear right justified, on input they may appear anywhere in the field you have delimited. Blanks (by default) are considered not to exist, for the purpose of the value read, although they do contribute to the field width. Apart from the digits 0 to 9, the only other characters which may appear in an integer field are – and +.

```
        READ(UNIT=*,FMT=100) I,J,K
100     FORMAT(3I4)
```

with the following values:–

```
2  –4 0+ 21
```

would result in the values 2, – 40 and 21 being assigned to I, J and K respectively.

Reals, the F and E formats

Real numbers may be input using either the E or F format, whether or not the E descriptor is present in the field. Again, we define a width, and, as with output, the number of places after the point:–

```
        F10.4
        E12.3
        F6.0
        E10.0
```

However, if the point is already present in the value being input, this overrides the definition in the format. Again, blanks are treated as null values.

```
        READ(UNIT=*,FMT=100) A,B,C
100     FORMAT(F10.6,E12.6,F6.0)
```

with

```
1234567890  14          4  .
```

results in A taking the value 1234.56789, B taking the value 14.0, and C the value 4.

The absence of the *E* in the field for B has no adverse affect. As a general rule, it is best to retain the decimal point with real numbers, just as a precaution.

Sometimes it is difficult to line up fields properly, and the first sign of trouble may be finding two decimal points in the one field, which will generate an error message, e.g. consider:–

```
        READ(UNIT=*,FMT=101) A,B,C,X,Y,Z
101     FORMAT(6F5.2)
```

If this was in a loop to read the following values:–

```
2.0  3.0  13.0 6.1  0.9  0.2
12.0  62.  9. –4.2 2.9 –3.7
```

The second time the READ was used, you would get an error (can you see why and where?)

An exponential format number (which may be read in F or E formats) can take a number of different forms. The most obvious is the explicit form:–

```
–1.2E–4
```

where all the components of the value are present — the significant digits to the left of the E, the E itself, and the exponent to the right. We can drop almost any two of these three components, and therefore:–

```
-1.2
-1.2E
-1.2-4
-4
```

are all valid values. Only the first two are interpreted as the same numerical value, and just giving the exponent part would be interpreted by the format as just giving the significant digits. If the exponent is to be given, there must be some significant digits also. It is not even enough to give the E and assume that the program will interpret this as 10 to the power *exponent* :–

```
E–4
```

is not an acceptable exponential format value, although:–

```
1E–4
```

would be.

There are opportunities for confusion with E formats.

```
        READ(UNIT=*,FMT=102) X,Y
102 FORMAT(2E10.3)
```

with:–

```
10.23 –2
```

This would be interpreted as X taking the value 10.23E-2 and Y taking the value 0.0, while with

```
102 FORMAT(2F8.3)
```

X would be 10.23, and Y would be –2.0.

Although the decimal point may also be dropped, this may generate confusion too. While:–

```
4E3
45
45E–4
45–4
```

are all valid forms, if an E format is used, a special conversion takes place. A format, like E10.8, when used with integral significant digits (no decimal point), uses the 8 as a *negative* power of 10 scaling, e.g.

```
3267E05
```

converts to

```
3267*10**–8*10**5
```

or

```
3267*10**3
```

or

```
3.267.
```

Therefore, the interpretation of, say, 136, read in E format, would depend on the format used:–

Value	Format		Interpretation
136	E10.0		136.0
136	E10.4		136.0*10**–4
		or	0.0136
136	E10.10		136.0*10**–10
		or	0.0000000136
136.	any above		136.0

One implication of all this is that the format you use to input a variable may not be suitable to output that same variable.

Blanks, nulls and zeros

You can control how Fortran treats blanks in input through two special format instructions, BN and BZ. BN is a shorthand form of *blanks become null,* that is, a blank is treated as if it was not there at all. BZ is therefore *blanks become zeros.*

As we have already seen, 1 4 (i.e. the two digits separated by a blank) read in I3 format would be read as 14; similarly, 14 (one-four-blank) is also 14 when the BN format is in operation. All of the blanks are ignored for the purposes of interpreting the number. They help to create the width of the number, but otherwise contribute nothing. This is the default, which will be in operation unless you specify otherwise.

The BZ descriptor turns blanks into zeros. Thus, 1 4 (one-blank-four) read in I3 format is 104, and 14 (one-four-blank) is 140.

There is one place where we must be very careful with the use of the BZ format — when using exponent format input. Consider:–

```
5.321E+02
```

read in (BZ,E10.3) format. We have specified a field which is ten characters wide, therefore the blank in column 10, which follows the E+02, is read as a zero, making this E+020. This is probably not what was required.

Characters

When characters are read in, it is sufficient to use the A format, with no explicit mention of the size of the character string, since this size (or length) is determined in the program by the CHARACTER declaration. This implies that any *extra* characters would not be read in. You may however read in less:–

```
      CHARACTER*10 LIST
      .
      .
      READ(UNIT=5,FMT=100)LIST
100   FORMAT(A1)
```

would read only the first character of the input. The remaining 9 characters of LIST would be set to blank.

The notion of blanks as nulls or zeros has no meaning for characters. The blank is a legitimate character, and is treated as meaningful, completely distinct from the notion of a null or a zero.

Skipping spaces and lines

The X format is also useful for input. There may be fields in your data which you do not wish to read. These are easily omitted by the X format:–

```
      READ(UNIT=5,FMT=100) A,B
100   FORMAT(F10.4,10X,F8.3)
```

Similarly, you can *jump over* or ignore entire records, by using the oblique. Do note however, that

```
      READ(UNIT=5,FMT=100) A,B
100   FORMAT(F10.4/F10.4)
```

would read A from one line (or record) and B from the next. To omit a record between A and B, the format would need to be:–

```
100   FORMAT(F10.4//F10.4)
```

Another way to skip over a record is:–

```
      READ(UNIT=5,FMT=100)
100   FORMAT()
```

with no variable name at all.

Reading

As you have seen already, reading, or the input of information, is accomplished through the READ statement. We have used:–

```
      READ *,X,Y
```

for list directed input from the terminal, and:–

```
      READ(UNIT=5,FMT=100) X,Y
```

for formatted input also from the terminal. These forms may be expanded to

```
      READ(UNIT=*,FMT=*) X,Y
```

or

```
      READ(UNIT=*,FMT=100) X,Y
```

for input from the terminal, or to

```
      READ(UNIT=5,FMT=*) X,Y
```

or

```
      READ(UNIT=5,FMT=100) X,Y
```

when we wish to associate the READ statement with a particular unit number (or format label, for formatted input). As with the WRITE statement, these last two READ statements may be abbreviated to

```
      READ(5,*) X,Y
```

and

```
READ(5,100) X,Y
```

File manipulation again

The OPEN and CLOSE statements are also relevant to files which are used as input, and they may be used in the same ways. Besides introducing the notion of manipulating lots of files, the OPEN statement allows you to change the default for the treatment of blanks. The default is to treat blanks as null, but the statement BLANK=’ZERO’ changes the default to treat blanks as zeros. There are other parameters on the OPEN, which are considered elsewhere.

Once you have OPENed a file, you may not issue another OPEN for the same file until it has been CLOSEd, except in the case of the BLANK= parameter. You may change the default back again with:–

```
OPEN(UNIT=10,FILE=’EXAMPL’)
READ(UNIT=10,FMT=100) A,B
.
.
.
OPEN(UNIT=10,FILE=’EXAMPL’,BLANK=’ZERO’)
READ(UNIT=10,FMT=100) A,B
```

This implies that, within the same input file, you may treat some records as blank for null, and some as blank for zero. This sounds very dangerous, and would be better done by manipulating individual formats if it had to be done at all.

Given that you may write a file, you may also *rewind* it, in order to get back to the beginning. The syntax is similar to the other commands:–

```
REWIND(UNIT=1)
```

This often comes in useful as a way of providing backing storage, where intermediate data can be stored on file and then used at a later part of the processing.

The notion of records in Fortran input and output has been introduced. If you are confident in your understanding of this ambiguous and nebulous concept, you can *backspace* through a file, using the statement

```
BACKSPACE(UNIT=1)
```

which moves back over a single record on the designated file. There is no point in trying to BACKSPACE or REWIND input, if that input is the terminal.

ERR and END

In discussing some aspects of input, it has been pointed out that errors may be made. Where such errors are noticed, in the sense that something illegal is being attempted, there are two options

- print a diagnostic message, and allow correction of the mistake
- print a diagnostic message, and terminate the program

The only time that the first makes sense is when you are interacting with a program at a terminal. Some Fortran implementations provide correction facilities in a case like this, but most do not.

This latter case may not be desirable, and you have a mechanism through a parameter on the READ statement to trap this.

```
        READ(UNIT=5,FMT=102,ERR=200) X,Y
```

would allow *faulty* data to be trapped. The keyword ERR= directs the program to label 200, where some sort of processing might occur, e.g.

```
        .
        NUM=0
1       NUM=NUM+1
        READ(UNIT=5,FMT=102,ERR=200) A,B
        .
        .
        .
200     WRITE(UNIT=6,FMT=103) NUM
103     FORMAT(' ERROR IN DATA INPUT, AT RECORD ',I4)
.
        .
```

While this does not guarantee correct values in A and B, like having decimal points in integer fields, or two decimal points in real fields (or before the sign), you might inadvertently try to read characters in I, F or E formats. Of course, reading numbers in A format would go by unnoticed.

Very often we do not know exactly how much data is to be read in. Unless you do something about it, reading beyond the end of the data on a file will generate an error, a fatal error. In some cases this is probably a good thing, but another parameter on the READ allows it to be done elegantly.

```
        READ(UNIT=5,FMT=100,END=101) LIST
```

As with the ERR= parameter, this directs the program to a given label in the event of hitting the end of the data input file (in this case, unit 5). Both the END= and ERR= belong to a *special* class of statements, those which are processed on discovering an error condition. This restricts their use to particular situations, and does not necessarily destroy the structure of the program.

Summary

• Values may be read in from the terminal or from another file through fixed formats.

• Much of the structure of input format statements is very similar to that of the output formats. Broadly speaking, data written out in a particular format may be read in by the same format. However, there is greater flexibility, and quite a variety of forms can be accepted on input.

• A key distinction to make is the interpretation of blanks, as either nulls or zeros; alternative interpretations can radically alter the structure of the input data

• Fortran allows file names to be associated with unit numbers through the OPEN statement. This statement allows control of the interpretation of blank, although this can also be done through the BN and BZ formats.

• The READ statement, besides allowing the input to come from a particular file, also allows checks to be made on the data, through the ERR= parameter, and checks for the *end of data* condition through the END=.

• Files may also be manipulated through REWIND and BACKSPACE.

Problems

1. Write a program that will read in two reals and one integer, using

```
FORMAT(F7.3,I4,F4.1)
```

and that, in one instance treats blanks as zeros, and in the second treats blanks as nulls. Use PRINT *, to print the numbers out immediately after reading them in. What do you notice? Can you think of instances where it is necessary to use one rather than the other?

2. Write a program to read in and write out a real number using

```
FORMAT(F7.2)
```

What is the largest number that you can read in and write out with this format? What is the largest negative number that you can read in and write out with this format? What is the smallest number, other than zero, that can be read in and written out?

3. Rewrite two of the earlier programs that used READ,* and PRINT,* to use FORMAT statements.

4. Write a program to read the file created by either the temperature conversion program or the litres and pints conversion program. Make sure that the programs ignore the line–printer control characters, and any header and title

information. This kind of problem is very common in programming (writing a program to read and possibly manipulate data created by another program).

5. Use the OPEN, REWIND, READ and WRITE statements to input a value (or values) as a character string, write this to a file, rewind the file, read in the values again, this time as real variables with blanks treated as null, then repeat with blanks as zeros.

6. Demonstrate that input and output formats are not symmetric — i.e. what goes in does not necessarily come out.

7. Can you suggest why Fortran treats blanks as null rather than zero?

8. What happens at your terminal when you enter faulty data, inappropriate for the formats specified? Does the operating system intercept the data, or can you use the ERR = escape route?

12

Making decisions (1)

The more alternatives, the more difficult the choice

Abbe d'Allainval, Title of comedy

Aims

The aims of this chapter are to introduce:–

- selection between various courses of action as part of the problem solution
- the concepts and statements in Fortran needed to support the above. In particular:–
    - logical expressions
    - logical operators
    - a *block* of statements
    - several *blocks* of statements

Selection between courses of action

In most problems you need to chose between various courses of action e.g.

- if overdrawn, then do not draw money out of the bank
- if Monday, Tuesday, Wednesday, Thursday or Friday, then go to work
- if Saturday, then go to watch Queens Park Rangers
- if Sunday, then lie in bed for another two hours

As most problems involve selection between two or more courses of action it is necessary to have the concepts to support this in a programming language. Fortran has a variety of selection mechanisms, some of which are introduced.

The BLOCK IF statement.

The following short example illustrates the main ideas:–

```
. wake up
.
. check the date and time
IF (TODAY.EQ.SUNDAY) THEN
        .
        . lie in bed for another two hours
        .
        .
ENDIF
.
. get up
. make breakfast
```

If today is Sunday then the block of statements between the IF and the ENDIF is executed. After this block has been executed the program continues with the statements after the ENDIF. If today is not Sunday the program continues with the statements after the ENDIF immediately. This means that the statements after the ENDIF are executed whether or not the expression is true.

The general form is:–

```
IF (Logical expression) THEN
        .
        .
        Block of statements
        .
        .
ENDIF
```

The logical expression is an expression that will be either true or false, hence its name. Some examples of logical expressions are given below:–

```
(ALPHA.GT.10.1)
```

Test if ALPHA 10.1

```
(BALANC.LT.0.0)
```

Test if overdrawn

```
((TODAY.EQ.SATDAY).OR.(TODAY.EQ.SUNDAY))
```

Test if today is Saturday or Sunday

```
((ACTUAL–CALC).LT.0.000001)
```

Test if ACTUAL minus CALC less than 0.000001

Fortran has the following relational operators:–

Operator	Meaning
.EQ.	Equal
.NE.	Not equal
.GE.	Greater than or equal
.LE.	Less than or equal
.LT.	Less than
.GT.	Greater than

and the following logical operators:–

Operator	Meaning
.AND.	and
.OR.	or
.NOT.	not

The first six should be self-explanatory. They enable expressions or variables to be compared and tested. The last three enable the construction of quite complex comparisons, involving more than one test; in the example given earlier there was a test to see whether today was Saturday or Sunday.

One special case of the IF statement may be useful. From time to time there may only be one statement to execute in a BLOCK IF:–

```
IF(MONTH.EQ.2)THEN
    NDAYS=28
ENDIF
```

In these circumstances, it is possible to compress the statements to a single logical if:–

```
IF(MONTH.EQ.2)NDAYS=28
```

This has exactly the same effect. Whichever form you use is a matter of taste – though the general form has the advantage of flexibility.

Note that some symbols available on the keyboard e.g.

```
>
>=
<
<=
<>
=
```

are not acceptable as a shorthand way of denoting the relational operators.

Use of logical expressions and logical variables (something not mentioned so far) are covered again in a later chapter on additional data types.

The ‘IF *expression* THEN *statements* ENDIF’ is called a BLOCK IF construct. There is a simple extension to this provided by the ELSE statement. Consider the following example:–

```
IF (BALNCE.GE.0.0) THEN
        .
        . draw money out of the bank
        .
        .
ELSE
        .
        . borrow money from a friend
        .
ENDIF
.
. Buy a round of drinks
.
```

In this instance, one or other of the blocks will be executed. Then execution will continue with the statements after the ENDIF statement (in this case *buy a round*).

There is yet another extension to the BLOCK IF which allows ELSEIF statement. Consider the following example:–

```
IF (TODAY.EQ.MONDAY) THEN
    .
ELSEIF (TODAY.EQ.TUSDAY) THEN
    .
ELSEIF (TODAY.EQ.WEDDAY) THEN
    .
ELSEIF (TODAY.EQ.THRDAY) THEN
    .
ELSEIF (TODAY.EQ.FRIDAY) THEN
    .
ELSEIF (TODAY.EQ.SATDAY) THEN
    .
ELSEIF (TODAY.EQ.SUNDAY) THEN
    .
ELSE
    there has been an error. The variable TODAY has
    taken on an illegal value.
ENDIF
```

Note that, as soon as one of the logical expressions is true, the rest of the test is skipped, and execution continues with the statements after the ENDIF. This implies that a construction like:–

```
IF(I.LT.2)THEN
    .
    .
ELSEIF(I.LT.1)THEN
    .
    .
ELSE
    .
ENDIF
```

is inappropriate. If I is less than 2, the latter condition will never be tested. The ELSE statement has been used here to aid in trapping errors or exceptions. This is recommended practice. A very common error in programming is to assume that the data is in certain well-specified ranges. The program then fails when the data goes outside this range. It makes no sense to have a day other than Monday, Tuesday, Wednesday, Thursday, Friday, Saturday or Sunday.

Examples

• This program is straightforward, with a simple structure. The roots of the quadratic are either *real*, *equal and real*, or *complex* depending on the magnitude of the term B ** 2 – 4 * A * C. The program tests for this term being greater than and less than zero, it assumes that the only other case is equality to zero (from the mechanics of a computer, floating point equality is rare, but, we are safe in this instance).

```
      PROGRAM QROOTS
      REAL A,B,C,TERM,A2,ROOT1,ROOT2
C
C   A B AND C ARE THE COEFFICIENTS OF THE TERMS
C   A*X**2+B*X+C
C   FIND THE ROOTS OF THE QUADRATIC, ROOT1 AND ROOT2
C
      PRINT*,' GIVE THE COEFFICIENTS A, B AND C'
      READ*,A,B,C
      TERM = B*B – 4.*A*C
      A2 = A*2.
C IF TERM < 0, ROOTS ARE COMPLEX
C IF TERM = 0, ROOTS ARE EQUAL
C IF TERM > 0, ROOTS ARE REAL AND DIFFERENT
      IF(TERM.LT.0.0)THEN
         PRINT*,' ROOTS ARE COMPLEX'
      ELSEIF(TERM.GT.0.0)THEN
         TERM = TERM**0.5
         ROOT1 = (–B+TERM)/A2
         ROOT2 = (–B–TERM)/A2
         PRINT*,' ROOTS ARE ',ROOT1,' AND ',ROOT2
      ELSE
         ROOT1 = –B/A2
         PRINT*,' ROOTS ARE EQUAL, AT ',ROOT1
      ENDIF
      END
```

• This next example is also straightforward. It demonstrates that, even if the conditions on the IF statement are involved, the overall structure is easy to determine. The comments and the names given to variables should make the program self-explanatory. Note the use of integer division to identify leap years.

```
      PROGRAM DATE
      INTEGER YEAR,N,MONTH,DAY,T
C
C  CALCULATES DAY AND MONTH FROM YEAR AND DAY-WITHIN-YEAR
C  T IS AN OFFSET TO ACCOUNT FOR LEAP YEARS
C
      PRINT*,' YEAR, FOLLOWED BY DAY WITHIN YEAR'
      READ*,YEAR,N
C     CHECKING FOR ORDINARY LEAP YEARS
      IF(((YEAR/4)*4).EQ.YEAR)THEN
        T=1
      ELSE
        T=0
      ENDIF
C     CHECKING FOR LEAP YEARS AT CENTURIES
      IF   (((YEAR/400)*400.EQ.YEAR)
     + .OR.((YEAR/100)*100.EQ.YEAR))THEN
        T=T
      ELSE
        T=0
      ENDIF
C      ACCOUNTING FOR FEBRUARY
      IF(N.GT.(59+T))THEN
        DAY=N+2-T
      ELSE
        DAY=N
      ENDIF
      MONTH=(DAY+91)*100/3055
      DAY=(DAY+91)-(MONTH*3055)/100
      MONTH=MONTH-2
      PRINT*,' CALENDAR DATE IS ',DAY,MONTH,YEAR
      END
```

Summary

• Decisions are a key part of problem solving, and of Fortran.

• Decisions are made on the basis of an IF statement, where some condition is evaluated as either true or false, and then a particular course of action is followed.

• The IF construct can be expanded quite elegantly into the IF-THEN-ELSE-ENDIF type of structure (the Block If), where the alternatives are grouped in a kind of *parenthetical* structure.

• Besides the ELSE, another statement, the ELSEIF may be used.

Problems

The physical world has many examples where processes require some threshold to be overcome before they begin operation: critical mass in nuclear reactions, a given slope to be exceeded before friction is overcome, and so on. Unfortunately, most of these sorts of calculations become rather complex and not really appropriate here. The following problem trys to restrict the range of calculation, whilst illustrating the possibilities of decision making.

1. If a cubic equation is expressed as

$$z^3 + a_2z^2 + a_1z + a_0 = 0$$

and we let

$$q = a_1/3 - (a_2\ a_2\ )/\ 9$$

and

$$r = (a_1a_2\text{-}3a_0)/6\text{-}(a_2a_2a_2)/27$$

we can determine the nature of the roots as follows:

$q^3 + r^2 > 0$; one real root and a pair of complex;
$q^3 + r^2 = 0$; all roots real, and at least two equal;
$q^3 + r^2 < 0$; all roots real;

Incorporate this into a suitable program, to determine the nature of the roots of a cubic from suitable input.

2. The form of breaking waves on beaches is a continuum, but for convenience we commonly recognise three major types: surging, plunging and spilling. These may be classified empirically by reference to the wave period, T (seconds), the breaker wave height, H_b (metres), and the beach slope, m. These three variables are combined into a single parameter, B, where

$$B = H_b/(gmT^2)$$

g is the gravitational constant (981 cm sec^{-2}). If B is less than .003, the breakers are surging; if B is greater than 0.068, they are spilling, and between these values, plunging breakers are observed.

(i) On the east coast of New Zealand, the normal pattern of waves is swell waves, with wave heights of 1 to 2 metres, and wave periods of 10 to 15 seconds. During storms, the wave period is generally shorter, say 6 to 8 seconds, and the wave heights higher, 3 to 5 metres. The beach slope may be taken as about 0.1. What changes occur in breaker characteristics as a storm builds up?

(ii) Similarly, many beaches have a concave profile. The lower beach generally has a very low slope, say less than 1 degree (m=0.018), but towards the high tide mark, the slope increases dramatically, to say 10 degrees or more (m=0.18). What changes in wave type will be observed as the tide comes in?

3. Personal taxation is usually structured in the following way:–

no taxation on the first m_0 units of income;
taxation at t_1% on the next m_1 units;
taxation at t_2% on the next m_2 units;

taxation at t_3% on anything above.

For some reason, this is termed *progressive* taxation. Write a generalised program to determine net income after tax deductions. Write out the gross income, the deductions and the net income. You will have to make some realistic estimates of the tax thresholds m_i and the taxation levels t_i. You could use this sort of model to find out how sensitive revenue from taxation was in relation to cosmetic changes in thresholds and tax rates.

4. The specific heat capacity of water is 2009 J kg^{-1} K^{-1}; the specific latent heat of fusion (ice/water) is 335 kJ kg^{-1}, and the specific latent heat of vaporization (water/steam) is 2500 kJ kg^{-1}. Assume that the specific heat capacity of ice and steam are identical to that of water. Write a program which will read in two temperatures, and will calculate the energy required to raise (or lower) ice, water or steam at the first temperature, to ice, water or steam at the second. Take the freezing point of water as 273 K, and its boiling point as 373 K. For those happier with Celsius, 0^o C is 273 K, while 100^o c is 373 K. One calorie is 4.1868 J, and for the truly atavistic, 1 BTU is 1055 J (approximately).

13

Functions

I can call spirits from the vasty deep.
Why so can I, or so can any man; but will they come when you do call for them?

William Shakespeare, 'King Henry IV, part 1'

Aims

The aims of this chapter are:–

- to introduce system supplied functions
- to extend to user defined functions
- to extend to statement functions

Introduction

Fortran provides a large number of functions, chiefly for common mathematical evaluations. They are used in a straightforward way. If we take the common trigonometric functions, sine, cosine and tangent, the appropriate values may be calculated quite simply by

```
X=SIN(Y)
Z=COS(Y)
A=TAN(Y)
```

This is in rather the same way that we might say that X is a function of Y, or X is sine Y. Note that the argument, Y, is in *radians* not *degrees*. These functions are called *intrinsic functions*. A selection is given here:–

Function	Action	Example
INT	conversion to integer	J=INT(X)
REAL	conversion to real	X=REAL(J)
ABS	absolute value	X=ABS(X)
MOD	remaindering	I=MOD(K,L)
	remainder when I divided by J	
MAX	maximum value	X=MAX(A,B,C,D)
	(at least 2 arguments)	I=MAX(K,L)
MIN	minimum value	X=MIN(A,B,C,D)
	(at least 2 arguments)	I=MIN(K,L)
SQRT	square root	X=SQRT(Y)
EXP	exponentiation	Y=EXP(X)
LOG	natural logarithm	X=LOG(Y)
LOG10	common logarithm	X=LOG10(Y)
SIN	sine	X=SIN(Y)
COS	cosine	X=COS(Y)
TAN	tangent	X=TAN(Y)
ASIN	arcsine	Y=ASIN(X)
ACOS	arccosine	Y=ACOS(X)
ATAN	arctangent	Y=ATAN(X)
ATAN2	arctangent(a/b)	Y=ATAN2(A,B)

A complete list is given in Appendix E.

Note that some of these functions can take *either* real or integer arguments. These are special *generic* type, which means that the type of the result is determined by the type of the arguments.

You should not use variables which have the same name as the intrinsic functions.

There are one or two other key points to note. Some of the intrinsic functions have multiple arguments, e.g. MIN and MAX; these arguments must all be of the same type.

You may also replace arguments for functions by expressions, e.g.

```
      X = LOG(2.0)
```

or

```
      X = LOG(ABS(Y))
```

or

```
      X = LOG(ABS(Y)+Z/2.0)
```

Examples

This example uses only one function, the MOD (or modulus). It is used several times, helping to emphasise the usefulness of a convenient, easily referenced function. The program calculates the date of Easter for a given year. It is derived from an algorithm by Knuth, who also gives a fuller discussion of its importance of its algorithm. He concludes that the calculation of Easter was a key factor in keeping arithmetic alive during the Middle Ages in Europe. Note, that determination of the Eastern churches' Easter requires a different algorithm.

```
      PROGRAM EASTER
      INTEGER YEAR,METCYC,CENTRY,ERROR1,ERROR2,DAY
      INTEGER EPACT,LUNA
C A PROGRAM TO CALCULATE THE DATE OF EASTER
      PRINT *,' INPUT THE YEAR FOR WHICH EASTER'
      PRINT *,' IS TO BE CALCULATED'
      PRINT *,' ENTER THE WHOLE YEAR, E.G. 1978 '
      READ *,YEAR
C CALCULATING THE YEAR IN THE 19 YEAR METONIC CYCLE-METCYC
      METCYC = MOD(YEAR,19)+1
      IF(YEAR.LE.1582)THEN
         DAY = (5*YEAR)/4
         EPACT = MOD(11*METCYC-4,30)+1
      ELSE
C     CALCULATING THE CENTURY-CENTRY
         CENTRY = (YEAR/100)+1
C     ACCOUNTING FOR ARITHMETIC INACCURACIES
C     IGNORES LEAP YEARS ETC.
         ERROR1 = (3*CENTRY/4)-12
         ERROR2 = ((8*CENTRY+5)/25)-5
```

```
C       LOCATING SUNDAY
             DAY = (5*YEAR/4)-ERROR1-10
C       LOCATING THE EPACT(FULL MOON)
             EPACT = MOD(11*METCYC+20+ERROR2-ERROR1,30)
             IF(EPACT.LT.0)EPACT=30+EPACT
             IF((EPACT.EQ.25.AND.METCYC.GT.11).OR.EPACT.EQ.24)THEN
                 EPACT=EPACT+1
             ENDIF
         ENDIF
C     FINDING THE FULL MOON
         LUNA=44-EPACT
         IF(LUNA.LT.21)THEN
             LUNA=LUNA+30
         ENDIF
C     LOCATING EASTER SUNDAY
         LUNA=LUNA+7-(MOD(DAY+LUNA,7))
C     LOCATING THE CORRECT MONTH
         IF(LUNA.GT.31)THEN
             LUNA = LUNA – 31
             PRINT *,' FOR THE YEAR ',YEAR,
             PRINT *,' EASTER FALLS ON APRIL ',LUNA
         ELSE
             PRINT *,' FOR THE YEAR ',YEAR,
             PRINT *,' EASTER FALLS ON MARCH ',LUNA
         END
```

As well as noting the use of the MOD generic function in this program, it is also worth noting the structure of the decisions. They are *nested*, rather like the nested DO loops we met earlier. Note, however, that each IF block requires its own ENDIF.

Reasons for functions

What kinds of reasoning lead to the adoption of functions in programs?

Duplication Very often we wish to do the same sort of thing repeatedly in a program. For example, we may wish to solve sets of simultaneous equations, add matrices together, or find the minimum and maximum value in a set of data. Clearly, every time we wish to do this, we could include the appropriate bits of program, but this may involve us in lots of rather boring duplication of instructions. Not only do we have to include it lots of times, but the poor compiler has to examine it lots of times too. You shouldn't feel too bad about the compiler, but it does seem ludicrous to risk making errors in the duplication — after all, any statement labels will have to be changed, and it is generally when such changes are made that errors are introduced. So one reason for adopting functions, or *sub- programs* is to avoid needless, and potentially dangerous, duplication. This also has the effect of saving space.

Modularity In breaking logically self-contained and thus distinct modules or segments (solving sets of simultaneous equations, etc.), we are imposing a natural structure on the problem. We have already discussed problem solving, and one key element is to reduce an apparently unmanageable problem to a series of manageable chunks. As long as we can actually specify the steps, we have a chance to solve the overall problem. Sub-programs assist in achieving modularity, and can give each chunk a separate identity. Thus it is easier to visualise the problem and its solution.

Extension We need functions in order to extend the range of operations available in Fortran. For example, Fortran 77 has no operators for vectors and matrices — to do simple arithmetic on such structures we have to write a function.

Brevity There is perhaps one other guideline to offer before considering sub-programs in more detail. The shorter a unit is, the more likely you are to see the errors, either before you actually run the program, or later, when you are trying to understand why it failed. Like all rules, this is not infallible, but it is best not to make the sub-programs too elaborate.

Supplying your own functions

There are two stages here, firstly to define the function and secondly to reference or use it. The following defines a function:–

```
REAL FUNCTION FUN(X,Y,Z)
REAL X,Y,Z
FUN = X*Y**Z
END
```

where X is a local variable, FUN is function name, which obeys all the conventions regarding type and length, and A,B,C are arguments. To use this function, you reference or call it with a form like:–

```
V1 = FUN(A,B,C)
```

A complete program including this function is given below.

```
PROGRAM TRIAL
REAL A,B,C,V1
.
V1 = FUN(A,B,C)
.
END

REAL FUNCTION FUN(X,Y,Z)
FUN = X*Y**Z
END
```

The function has two important feature which distinguishes it from the PROGRAM segment:–

- the type of the function — in this case real. Functions return values, and the values returned have to be of a specific type.
- the *arguments* — in this case X,Y,Z. Note that, in the call, we have three arguments, and so too in the FUNCTION statement, and that the arguments are matched in order. There must be a one-to-one correspondence between the arguments, including their type and whether they are arrays, vectors, or simple variables.

Functions are treated by the compiler as completely separate entities. They will occur before or after (but never within) other programs or sub-programs, and when they are referenced, the flow of control will pass to them. At the end of the sub-program, control should usually be passed back to the calling routine. The END statement in the function terminates the action of that function, and the next statement to be executed will be the next in the calling routine. Consider the following example:–

```
        PROGRAM FRONT
        INTEGER FACT,I,J
        DO 1 I=–2,10
            J=FACT(I)
            PRINT 100,I,J
1       CONTINUE
100     FORMAT(1X,I4,' FACTORIAL IS ',I10)
        END

        INTEGER FUNCTION FACT(N)
        INTEGER N,I
        FACT=1
C
C THERE ARE THREE CASES.
C       1) N > 1                FACTORIAL EVALUATED
C       2) N = 0 OR N = 1   FACTORIAL IS 1
C       3) N < 0                FACTORIAL ILLEGAL
C
        IF(N.LT.0)THEN
            PRINT *,' NEGATIVE VALUE FOR FACTORIAL'
            PRINT *,' NOT DEFINED'
            FACT = 0
        ELSE
            DO 1 I = 2,N
                FACT = FACT*I
1           CONTINUE
        ENDIF
        END
```

There is another important feature. In a function called FACT, somewhere there must be a variable FACT appearing on the left hand side of an equals sign. Note that the type of FACT determines the type of the returned value.

What restrictions have been forced on us? Primarily, we can get only one answer, a simple variable, returned as the result of setting something equal to that function name (i.e. an explicit reference). Imagine that we wished to find the maximum and minimum of a vector of data. Solving this through functions actually requires two functions, e.g.

```
      XMIN = VMIN(X,N)
      XMAX = VMAX(X,N)
```

where VMIN and VMAX are the functions to find the minimum and maximum, X is the vector of values and N is the number of values in X. If we look at the actual code to calculate the values:–

```
      REAL FUNCTION VMIN(V,N)  REAL FUNCTION VMAX(V,N)
      INTEGER I,N                  INTEGER I,N
      REAL V                       REAL V
      DIMENSION V(100)             DIMENSION V(100)
      VMIN = V(1)              VMAX = V(1)
      DO 1 I=2,N               DO 1 I=2,N
          IF(V(I).LT.VMIN)THEN     IF(V(I).GT.VMAX)THEN
              VMIN=V(I)                    VMAX=V(I)
          ENDIF                        ENDIF
1     CONTINUE                 1   CONTINUE
      END                          END
```

There is clearly some duplication, and in a later chapter we will look at ways of eliminating even this overlap.

There is another way of terminating the action of a function besides the END statement. This is done using the RETURN statement. In each of the examples above, a RETURN could have been inserted before the END statement. Equally well however, the RETURN could be placed at any other logically appropriate position. The examples above offer little scope for alternative positions for a RETURN, but in more complex functions, this may be appropriate. This flexibility stems from the possibility of regarding functions in two logically distinct ways:–

- as an action you want carried out; in which case you END the action;
- as a section of program code that you *jump* to and execute; in this case you RETURN to the calling routine.

It is considered good practice to have only one exit route from a sub-program. This is perhaps an over-zealous interpretation of the tenets of structured programming, since it is often necessary to indicate an error condition in the function; manipulating the structure in order to ensure a single exit at the END statement may impose a degree of perversity on the flow. Where RETURN is associated with an error condition, there can be little to criticise.

The notion of functions, returning a single value through the function name, would seem to preclude notification of errors. It is possible to return other values through the arguments. In other words, the arguments to the function may also be used to transfer information from the function to the calling sub-program, as well as the more conventional direction. In general terms, this may be discouraged, but from time to time it is a useful feature. Later, a better structure to encompass this possibility will be introduced.

Statement functions

The statement function is a very simplified form of the function. If it is possible to compress the calculation required into a single statement (which might of course take up several continuation lines), it may be expressed as a statement function.

Such a function would occur within a program segment, immediately before the first executable statement. This very simplified form may not reference an array name (although it could reference an array element), and, if character variables are passed to it, there can be no sub-string references (see Chapter 17).

Since the statement function is specified within a program segment, it may only be used within that segment, and cannot be referenced from any other functions or subroutines, unlike the intrinsic or other user-defined functions.

The following are examples of the statement function:–

```
      CUBRT(A)=A**(1./3.)

      IDAY(I,J,K)=3055*(J+2)/100–(J+10)/13*2-91
     1+(1–(I–1/4*4+3)/4+(I–1/100*100+99)/100
     2–(I–1/400*400+399)/400)*(J+10)/13+K


      AREA(ANG,B,C)=B*C*SIN(ANG)*0.5
```

or

```
      AREA(A,B,C)=((A+B+C)*(B+C–A*0.5)*(A+C–B*0.5)*
     1        (A+B–C*0.5))**0.5
```

The first of these statement functions, CUBRT, is self-explanatory. The second, IDAY, requires a little more comment. IDAY calculates the day of the year on which a particular date falls, given the year I, the month J (where January is 1, February is 2, and so on), and the the last pair of functions calculate the area of a triangle; the first from the included angle ANG, and the sides A and B; the second from the three sides A, B and C. This last function is rather clumsy, can you see why?

The statement function is used as follows:–

```
      PROGRAM FACTOR
      REAL RESULT,PI,EN,R
      PARAMETER (PI=3.14159265358)
C  STIRL CALCULATES AN APPROXIMATION TO N! FOR LARGE N
      STIRL(X)=SQRT(2.*PI)*X**(X+0.5)*EXP(-X)
      .
      .
      EN=10.
      R=7.
C  NUMBER OF POSSIBLE COMBINATIONS THAT CAN BE FORMED WHEN
C  R OBJECTS ARE SELECTED OUT OF A GROUP OF EN
C                 R!/N!(N-R)!
      RESULT=STIRL(EN)/STIRL(R)*STIRL(EN-R)
      .
      .
      END
```

Summary

• There are a large number of Fortran supplied functions (intrinsic functions) which extend the power and scope of the language. Some of these functions are of *generic* type, and can take several different types of argument. Others are restricted to a particular type of argument.

• When the intrinsic functions are inadequate, it is possible to write *user defined* functions. Besides expanding the scope of computation, such functions help in problem visualisation and logical subdivision, may reduce duplication, and generally help in avoiding programming errors.

• In addition to separately defined user functions, statement functions may be employed. These are single statements which are used within a program segment.

• Although the normal exit from a user defined function is through the END, other, *abnormal,* exits may be defined through the RETURN statement.

• Communication with a function is through the function name and the function arguments. The function *must* contain a reference to the function name on the left hand side of an assignment. Results may also be returned through the argument list.

Problems

1. In Chapter 10 there is a program which calculates calendar dates from year and day within year. The statement function IDAY in this chapter reverses this operation, to calculate day within year from calendar dates. Combine these two elements in order to test their equivalence.

2. Type in and test the factorial example given in the chapter. The explicit formula used for the evaluation of R!/N!(N-R)! is rather crude. Write a function which improves on it.

3. Type in and test either the minimum or maximum example function. You will need a program segment to use the function.

4. Improve on the statement function:–

```
      AREA(A,B,C)=((A+B+C)*(B+C-A*0.5)*(A+C-B*0.5)*
     1            (A+B-C*0.5))**0.5
```

for the area of a triangle, where A, B and C are the lengths of individual sides. This need not be a statement function; test it. You might consider the situation where the input is incorrect, and A, B and C could not represent a triangle.

5. Find out the action of the MOD function when one of the arguments is negative. Write your own modulus function to return only a positive remainder. Don’t call it MOD!

6. Create a table which gives the sines, cosines and tangents for 0 degrees to 90 degrees in 1 degree intervals. There are a few minor catches in this question.

14

Making decisions (2)

Wilt thou still go down to destruction

William Blake, ‘Jerusalem’

Aims

The aims of this chapter are:–

- to introduce two other control structures that can be used both in decision making and for the control of repetition
    - the *while loop*
    - the *repeat until* construct

Other control mechanisms

There are many problems that you will meet that cannot be solved with the control mechanisms introduced so far. The two mechanisms introduced in this chapter do not have a direct form, rather they have to be constructed from more primitive forms. The two high level mechanisms are often written as:–

- while (expression) do (block of statements)

and

- repeat (block of statements) until (expression)

You should now be familiar with the ideas of both a logical expression, and of a block of statements, so the above should pose no problems to you. Note that the *while* construct may never be executed, and the *repeat* construct will always be executed once. The *while loop* is implemented in Fortran as:–

```
label IF (logical expression) THEN
            .
            . block of statements
            .
            GOTO label
        ENDIF
```

The following example shows a complete program using this construct.

```
        PROGRAM FIND
C THIS PROGRAM PICKS UP THE FIRST OCCURRENCE
C OF A NUMBER IN A LIST.
C A SENTINEL IS USED, AND THE ARRAY IS 1 MORE
C THAN THE MAX SIZE OF THE LIST.
        DIMENSION A(101)
        INTEGER A,MARK
        INTEGER END,I
        READ (UNIT=1,FMT=*) MARK
        READ (UNIT=1,FMT=*) END
        READ(UNIT=1,FMT=*) (A(I),I=1,END)
        I=1
        A(END+1)=MARK
100     IF(MARK.NE.A(I))THEN
            I=I+1
            GOTO 100
        ENDIF
        IF(I.EQ.(END+1)) THEN
            PRINT*,' ITEM NOT IN LIST'
        ELSE
            PRINT*,' ITEM IS AT POSITION ',I
        ENDIF
        END
```

The *repeat until* construct can be written in Fortran as:–

```
label   CONTINUE
             .
             .
             .     Body of the loop
             .
             .
        IF (expression ) GOTO label
```

There are problems in most disciplines that require a numerical solution. The two main reasons for this are that either the problem can only be solved numerically, or that an analytic solution involves too much work. Solutions to this type of problem often require the use of the *repeat until* construct. The problem will typically require the repetition of a calculation until the answers from successive evaluations differ by some small amount, decided generally by the nature of the problem.

Here is a program extract to illustrate this:–

```
        PARAMETER(TOL=10E–6)
        .
10      CONTINUE
             .
             .
             CHANGE=
             .
        IF (CHANGE.GT.TOL) GO TO 10
        .
```

The value of the tolerance is set here to 10E–6.

Examples

The function ETOX illustrates one use of the *repeat until* construct. The function evaluates e**x. This may be written as:–

$$1 + x/1! + x^2/2! + x^3/3! \ldots$$

or

$$1 + \sum_{n=1}^{\infty} x^{n-1}/(n-1)!\ (x/n)$$

Every succeeding term is just the previous term multiplied by *x/n*. At some point the term *x/n* becomes very small, so that it is not sensibly different from zero, and successive terms add little to the value. The function therefore repeats the loop until *x/n* is smaller than the tolerance. The number of evaluations is not known beforehand, since this is dependent on *x*.

```
        REAL FUNCTION ETOX(X)
        REAL TERM,X,TOL
        INTEGER NTERM
        PARAMETER (TOL=0.001)
        ETOX=1.0
        TERM=1.0
        NTERM=0
1       CONTINUE
            NTERM=NTERM+1
            TERM=(X/NTERM)*TERM
            ETOX=ETOX+TERM
        IF(TERM.GT.TOL)GO TO 1
        END
```

Both types of loop are combined in this last example. The algorithm employed here finds the zero of a function. Essentially, it finds an interval in which the zero must lie; the evaluations on either side are of different sign. The *while loop* ensures that the evaluations are of different sign, by exploiting the knowledge that the incident wave height must be greater than the reformed wave height (to give the lower bound). The upper bound is found by experiment, making the interval bigger and bigger. Once the interval is found, its mean is used as a new potential bound. The zero must lie on one side or the other; in this fashion, the interval containing the zero becomes smaller and smaller, until it lies within some tolerance. This approach is rather plodding and unexciting, but is suitable for a wide range of problems.

This example is drawn from a situation where a wave breaks on an offshore reef or sand bar, and then reforms in the near-shore zone before breaking again on the coast. It is easier to observe the heights of the reformed waves reaching the coast than those incident to the terrace edge.

```
      PROGRAM BREAK
      REAL HI,HR,HLOW,HIGH,HALF,XL,XH,XM,D,TOL
      PARAMETER (TOL=10E-6)
C PROBLEM - FIND HI FROM EXPRESSION GIVEN IN FUNCTION F
      F(A,B,C)=A*(1.0-0.8*EXP(-0.6*C/A))-B
C HI IS INCIDENT WAVE HEIGHT              (C)
C HR IS REFORMED WAVE HEIGHT                 (B)
C D IS WATER DEPTH AT TERRACE EDGE           (A)
      PRINT*,' GIVE REFORMED WAVE HEIGHT, AND WATER DEPTH'
      READ*,HR,D
C
C FOR HLOW- LET HLOW=HR
C FOR HIGH- LET HIGH=HLOW*2.0
C
C CHECK THAT SIGNS OF FUNCTION RESULTS ARE DIFFERENT
C
      HLOW=HR
      HIGH=HLOW*2.0
      XL=F(HLOW,HR,D)
      XH=F(HIGH,HR,D)
C              BEGINNING OF WHILE
1     IF((XL*XH).GE.0.0) THEN
        HIGH=HIGH*2.0
        XH=F(HIGH,HR,D)
        GOTO 1
      ENDIF
C          BEGINNING OF REPEAT UNTIL
2     HALF=(HLOW+HIGH)*0.5
        XM=F(HALF,HR,D)
        IF((XL*XM).LT.0.0)THEN
          XH=XM
          HIGH=HALF
        ELSE
          XL=XM
          HLOW=HALF
        ENDIF
      IF(ABS(HIGH-HLOW).GT.TOL)GO TO 2
C              END OF REPEAT UNTIL
      PRINT*,' INCIDENT WAVE HEIGHT LIES BETWEEN'
      PRINT*,HLOW,' AND ',HIGH,' METRES'
      END
```

Summary

You have been introduced in this chapter to two more control structures. These are the:–

- the *while* construct

and the

- the *repeat until* construct

These two constructs, together with the BLOCK IF, and IF THEN ELSEIF are sufficient to solve a wide class of problems.

The *repeat until* and *while* are both made up from the more primitive IF and GOTO statements. These latter two statements can be used in a variety of ways. However, it is essential that you restrict yourself to a small set of well defined structures. Unrestricted use of IF and GOTO statements can lead to a program that looks like a bowl of spaghetti, where the GOTOs take you on a mystery tour. The action of the program rapidly becomes very difficult to work out. Once this has happened inserting new features, and correcting the program may well become impossible. This will not be apparent at the start of programming, but experience will teach you the hard way.

Problems

1. Rewrite the program for period of a pendulum. The new program should print out the length of the pendulum, and period for lengths of the pendulum from 0 to 100 cms in steps of 0.5 cms. The program should incorporate a function for the evaluation of the period.

2. Using functions, do the following:–

- Evaluate n! from n=0 to n=10
- Calculate 76 factorial.
- Now calculate (x**n)/n!, with x=13.2 and n=20.
- Now do it another way.

3. The program BREAK is taken from a real example. In the particular problem, the reformed wave height was 1 metre, and the water depth at the reef edge was 2 metres. What was the incident wave height? Rather than using an absolute value for the tolerance, it might be more realistic to use some value related to the reformed wave height. These heights are unlikely to be reported to better than about 5 per cent accuracy. Wave energy may be taken as proportional to wave height squared for this example. What is the reduction in wave energy as a result of breaking on the reef or bar, for this particular case.

4. What is the effect of using INT on negative real numbers? Write a program to demonstrate this.

5. How would you find the nearest integer to a real number? Now do it another way. Write a program to illustrate both methods. Make sure you test it for negative as well as positive values.

6. The function ETOX has been given in this chapter. The standard Fortran function EXP does the same job. Do they give the same answers? Curiously the Fortran standard does not specify how a *standard* function should be evaluated, or even how accurate it should be.

15

Error detection and correction

We know that the program is correct, because we designed it correctly.

M. A. Jackson, 'Principles of Program Design'

Aims

The aims of this chapter are:–

- to introduce some of the common ways that errors get into programs
- to look at some of the ways the computer system can help in the process of error detection and correction

Introduction

Errors are due to a wide variety of causes. They may include simple typing errors, incorrect use of certain statements, logic errors etc. The computer system can help you in a variety of ways in the error detection and correction process.

The computer system can help detect errors at two stages. These are:–

- At the compilation stage

  The methods at this stage involve the selection of *compiler options,* and the use of an up-to-date listing of the program.

- At the execution stage

  Again, the method generally depends on a choice of *compiler options,* but now there may be the opportunity to use another program, sometimes given the name *post-mortem dump,* and on some systems there is the possibility of using an *interactive debugger*.

Facilities like those described below should be available on most systems.

The compilation process

In the first chapter it was stated that one of the things that a *compiler* does is to take a program written in a high level language and produce a set of machine level instructions that can be executed by the hardware. In fact, this is only one of a possibly large number of things that a compiler can do. Let us now consider some of the other functions of a compiler, particularly those which may help in error detection and correction.

Compiler options

Error trace back

What this means is that when an execution error occurs, there will be code added to your program that will try to work back from where the error caused the program to blow up to where the error may have been generated. Note that an error can occur quite early on in a program and take a considerable while before it has a serious and noticeable effect.

Array checking

Cause array bounds to be checked. An array going out of bounds is one of the most common errors in Fortran programming. For example, if a program calculates the index for an array reference, in order to place information into that array, it is possible for the program to go outside the memory set aside for the array. This means that the program could be over-writing itself. This kind of error is not always trapped and diagnosed by Fortran compilers.

String checking

Turn on the checking of sub-string operations on character data. This kind of error can be very difficult to find sometimes.

Post-Mortem Dump

Switch on the post-mortem dump. If your program goes wrong at execution time another program will try and work out where your program went wrong. This option requires the compiler generated *symbol tables* to be available. Symbol tables are compiler generated files. They are used in a variety of ways at the compilation stage. They can also be of use at the execution stage by other programs. The contents of a symbol table may be all of the variables with a list of the variable attributes, e.g. real, integer etc. These tables enable the post-mortem dump program to come in after your program has gone wrong and give you useful information regarding the state of your program at the time it went wrong. Another file that needs to be available is the load or linker listing. This is a listing file containing information about where variables etc. are actually to be found in memory. When you reference an array, for example, there will be a storage location associated in the linker listing.

Debug

Make available a run-time de-bugging environment. This enables your program to be interrupted and values of variables to be printed out or even changed. The beauty of this option is that no changes to the program are necessary, you interact with your program *through* the de-bugging program. This facility is simple yet powerful, and can save considerable amounts of time when de-bugging large programs.

Listing options

The compiler generates a file, the contents of which are discussed in more detail below, called a listing file. This information can be written to the terminal or to a file. Whilst working at a terminal it is possible to write this information to a file and then send the file to be printed.

When the compiler gives extra information and diagnostics it refers to the source. Hence an up-to-date listing is essential. Most people do *not* generate an up-to-date source listing each time they compile However, if you are trying to find bugs in your program, it makes no sense at all not to work from an up-to-date listing, reflecting the program at the time it went wrong. Often you will make small changes which you think do not cause any problems.

However, changing programs tends to make them go wrong, and you should always work from an up-to-date listing when trying to find errors.

We would want the listing to contain at least

- the complete source
- a list of all variables functions etc with their type
- a cross reference of all variables and where they occur in the program.

It is also useful with large programs if the listing has page numbering, and that each function (or subroutine) starts on a new page — it makes the listing easier to use, and locate information.

Optimise off

When developing programs there is no point getting the compiler to optimise your program when there are going to be errors. Optimisation is considered more generally in Chapter 21.

Summary

This chapter has looked at the kinds of facilities you could expect to be available when debugging programs.

There will probably be debugging aids available on your machine that have direct counterparts to the ones mentioned above, and really it is up to you to see what your system has to offer in this area.

Problems

1. Find out what options there are when compiling your program. In what way are they similar to the ones mentioned in this chapter? In what ways are they different?

16

Complex, double precision and logical

A messenger yes/no semaphore
her black/white keys in/out whirl of morse
hoopooe signals salvation deviously

Nathaniel Tarn, The Laurel Tree

Aims

The aims of this chapter are:–

- to review the variable types already introduced: real, integer, character
- to examine the other variable types available in Fortran: double precision, complex and logical
- to introduce the concepts necessary to use logical expressions effectively; namely, logical variables, hierarchy of operations, the truth table

Introduction

Fortran recognises a variety of variable types. You have already encountered real, integer and character. Real variables are those which may take any numerical value, within the range of the machine, while integers may take only integral, or whole number values. Characters, as their name suggests, contain character information, which is not numerical at all. These three types of data store or encode their information differently, so that the actual representation of the information is quite different. Consider the following:–

```
REAL X
INTEGER I
CHARACTER*2 C
X=10.0
I=10
C='10'
```

Although X, I and C each contain a 10, the machine perceives these values rather differently. The following is an example of how one machine stores the above values. We use octal (base 8) numbers to express the underlying bit patterns. You don't need to understand the following completely, just appreciate that they are radically different.

CDC, 60 bit

Integer	10	=	00000000000000000012
Real	10.0	=	17235000000000000000
Character	10	=	34335555555555555555

There are 49 characters in the standard Fortran character set. You may, however, store any character that is available within the operating system character set. This can vary considerably. Some older machines (CDC) use 6 bits to represent characters. This means that you can only have 64 characters available, without having to resort to some more complicated scheme. Most machines now use eight bits to represent characters, and this makes easily available the two most common character sets, ASCII and EBCDIC. The ASCII character set is given in appendix A. On most machines that use ASCII, there are 95 printing characters, and on machines that use EBCDIC (IBM and Amdahl) you have even more! Why this difference? The Fortran character set is defined for forming variable names, for numerical information, and for the operators which are needed. However, any pattern which can be legitimately expressed on the machine may be legitimately stored into a character variable. There are no checks performed on the contents of such variables.

If you say nothing to the contrary, reals and integers will take their types from the first character of the variable name; integers begin with either I, J, K, L, M or N, while reals begin with any other alphabetic character. You can over-ride this implicit typing by declaring given variables to be REAL or INTEGER, as we have tried to do in the preceding chapters, e.g.

```
REAL A2,INDEX
INTEGER INCH,AGE
```

Any other variable type must be declared explicitly. In Fortran, up to six characters are allowed for variable names.

There is another way of over-riding the default typing, through the IMPLICIT declaration. This allows you to specify the type to be taken by all variable names *beginning* with a particular letter or range of letters:

```
IMPLICIT REAL(K-R)
IMPLICIT INTEGER (A)
IMPLICIT CHARACTER (C-E,X)
```

would make all variables beginning with K, L, M, N, O, P, Q and R real variables; those beginning A would be integer, while those whose names had the initial letter C, D, E and X would be characters. Elsewhere, default typing would still be in operation. The IMPLICIT statement may be used for the other variable types to be mentioned in this chapter.

Double Precision

The double precision variable type is an extension to the real variable type and reflects the fact that we are dealing with a digital machine which has only a limited precision. At the top end of the range of scientific machines (Cray and CDC) 64 and 60 bits are used respectively to represent reals. This means that 1.00 000 000 000 01 is different from 1.0. On a smaller word size machine (e.g. 48-bit, 32-bit, 16-bit) the two numbers would not be distinguishable. In order to accommodate some of these potential problems of different word size, and sometimes just to increase precision (necessary with certain algorithms), we can extend the number of bits used to represent a real number. Double precision on a CDC or Cray gives 120 or 128 bits respectively. This is rarely used, but is occasionally required. However, double precision used with a 32-bit machine would give similar accuracy to the 60-bit CDC word, or 64-bit Cray word. Double precision will be necessary on smaller word length machines in most applications. Note that double precision is only applicable to real numbers. There is no concept of double precision for integers.

In order to use double precision you must declare it explicitly:

```
DOUBLE PRECISION A1,A2
. . .
A1=A2**2
```

All the functions and operators which work with real will work with double precision, but, as always, be careful when you mix types in performing calculations.

Reading and writing double precision values is done in exactly the same way that you read and write any other real number — through the E and F formats.

Complex

This variable type reflects a change in the nature of the data – the COMPLEX data type, where we can store and manipulate complex variables. Unless you are an engineer, or one of various varieties of mathematician, you won't find this particularly useful. Complex numbers are defined as having a 'real' and 'imaginary' part; i.e.

$$a = x + iy$$

where i is the square root of –1.

They are used heavily to solve a limited range of problems in certain disciplines, and they are not supported in many programming languages as a base type. To use this variable type we have to write the number as two parts, the real and imaginary elements of the number, for example

```
COMPLEX U
U=(1.0,2.0)
```

represents the complex number 1+i2. Note that the complex number is enclosed in brackets. We can do arithmetic on variables like this, and most of the intrinsic functions like LOG, SIN, COS etc. accept complex data type as argument. However, note that any user-defined functions which return a complex would have to be of the form:

```
COMPLEX FUNCTION OMEGA(U)
```

or something similar. All the usual rules about mixing different variable types, like reals and integers, also apply to complex. Complex numbers are read in and written out in a similar way to real numbers, but with the provision that, for each single complex value, two format descriptors must be given. You may use either E or F formats (or indeed, mix them), as long as there are enough of them. Although you use brackets around the pairs of numbers in a program, these must not appear in any input, nor will they appear on the output.

Fortran has a number of functions which help to clarify the intent of *mixed mode* expressions. The functions REAL, DBLE, CMPLX and INT can be used to 'force' any variable to real, double precision, complex or integer type. Thus

```
INTEGER I
DOUBLE PRECISION A
I=1
A=DBLE(I)
```

will convert an integer variable or value into a double precision variable or value. In fact, in the extract above,

```
A=I
```

would have had the desired effect too. However it is generally regarded as good practice to use the first form, as it makes explicit exactly what is meant.

Where this set of functions becomes valuable is when we have more complicated expressions to evaluate, where we might be concerned that the arithmetic might be done in mixed type, with results which were not truly anticipated;

```
A=I/J
```

will give integer arithmetic in the division, while

```
A=REAL(I)/REAL(J)
```

would do the division in real arithmetic. Note however that the following will not do the I/J in real arithmetic.

```
A=REAL(I/J)
```

CMPLX is a little different, since, as we have already seen, it can take two arguments. When it does, they must both be of the same type, integer, real or double precision. When they are double precision, only the first ‘half’, i.e. the single precision real part is used. There is no such thing as double precision complex. When only one argument is present, it is assumed to be the ‘real’ part, and the imaginary part is set to zero.

Remember that INT always returns the ‘truncated’ part of the number, removing the parts after the decimal point. Thus we must think carefully about its effect on negative numbers.

Logical

Often we have situations where we need ON/OFF, TRUE/FALSE or YES/NO switches, and in such circumstances we can use LOGICAL type variables: e.g.

```
LOGICAL FLAG
```

Logicals may take only two possible values, as shown following

```
FLAG=.TRUE.
```

or

```
FLAG=.FALSE.
```

Note the full stops, which are essential. With a little thought you can see why they are needed. You will already have met some of the ideas associated with logical variables from IF statements.

```
IF(A.EQ.B) THEN
    .
ELSE
    .
ENDIF
```

The logical expression (A.EQ.B) returns a value *true* or *false*, which then determines the route to be followed; if the quantity is true, then we execute the next statement, else we take the other route.

Similarly, the following example is also legitimate:

```
LOGICAL ANSWER
ANSWER=.TRUE.
.
.
IF (ANSWER) THEN
    .
ELSE
    .
ENDIF
```

Again the expression IF (ANSWER) is evaluated; here the variable ANSWER has been set to .TRUE., and therefore the statements following the THEN are executed. Clearly, conventional arithmetic is inappropriate with logicals. What does 2 times true mean? (very true?). There are a number of special operators for logicals:

.NOT. which negates a logical value (i.e. changes *true* to *false* or vice versa)

.AND. logical union

.OR. logical intersection

To illustrate the use of these operators, consider the following program extract:

```
      LOGICAL A,B,C
      A=.TRUE.
      B=.NOT.A
C                             (B now has the value 'false')
      C=A.OR.B
C                             (C has the value 'true')
      C=A.AND.B
C                             (C now has the value 'false')
```

To gauge the effect of these operators on logicals, we can consult a truth table:–

X1	X2	.NOT.X1	X1.AND.X2	X1.OR.X2
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

As with arithmetic operators, there is an order of precedence associated with the logical operators.

.AND. is carried out before

.OR. and .NOT.

In dealing with logicals, the operations are carried out within a given level, from left to right. Any expressions in brackets would be dealt with first. The logical operators are a lower order of precedence to the arithmetic operators, i.e. they are carried out later. A more complete operator hierarchy is therefore:

expressions within brackets
exponentiation
multiplication/division
addition/subtraction
relational logical (.GT. .GE. .LT. .LE. .EQ. .NE.)
.AND.
.OR. and .NOT.

Although you can build up complicated expressions with mixtures of operators, these are often difficult to comprehend, and it is generally more straightforward to break 'big' expressions down into smaller ones, whose purpose is more readily appreciated.

Historically, logicals have not been in evidence extensively in Fortran programs, although clearly there are occasions on which they are of considerable use. Their use often aids considerably in making programs more modular and comprehensible. They can be used to make a complex section of code involving several choices much more transparent by the use of one logical function, with an appropriate name. Logicals may be used to control output, e.g.

```
        LOGICAL DEBUG
        .
        DEBUG=.TRUE.
        .
        IF(DEBUG) PRINT *,'LOTS OF PRINTOUT'
```

ensures that, while de-bugging a program you have more output. Then, when the program is 'correct', run with DEBUG=.FALSE.

Note that Fortran does try to protect you while you use logical variables. You cannot do this:

```
        LOGICAL UP, DOWN
        UP=DOWN+.FALSE.
```

or

```
        LOGICAL A2
        REAL OMEGA
        DIMENSION OMEGA(10)
        .
        A2=OMEGA(3)
```

The compiler will note that this is an error, and will not permit you to run the program. This is an example of *strong typing*, since only a limited number of predetermined operations are permitted. The real, integer, complex and double precision variable types are much more weakly typed (which helps to lead to the confusion inherent in mixing variable types in arithmetic assignments).

Since logicals may take only the values .TRUE. and .FALSE., the possibilities in reading and writing logical values are clearly limited. The L format allows logicals to be input and output. On input, if the first non-blank characters are either T or .T, the logical value .TRUE. is stored in the corresponding list item; if the first non-blank characters are F or .F, then .FALSE. is stored. (Note therefore that reading, say, TED and FAHR in an L4 format would be accept-able.) If the first non-blank character is not F, T, .F or .T, then an error message will be generated. On output, the value T or F is written out, right justified, with blanks (if appropriate). Thus,

```
        LOGICAL FLAG
        FLAG=.TRUE.
        PRINT *, FLAG, .NOT.FLAG
100     FORMAT(2L3)
```

would produce

```
T  F
```

at the terminal.

Assigning a logical variable to anything other than a .TRUE. or .FALSE. value in your program will result in errors. The ‘shorthand’ forms of .T, .F, F and T are not acceptable in the program.

Typing with functions

Most of the mathematical functions will take any type of argument (excluding logical and character), but there are cases where this is a little absurd. Integer is not an appropriate argument for some functions, and should not be used. Although a good compiler will tell you when you have made errors in typing, not all compilers are so helpful. It is best to try to develop a defensive style where you depend as little as possible on the compiler interpreting complicated or abstruse statements.

With user-defined functions we need to have the function name of the correct type (integer, real, double precision, complex, logical or character). In the case of real and integer we can let default typing take over, through the first character of the function name, but for the others we must use a structure like:

```
      LOGICAL FUNCTION NOMORE(A,B)
      REAL A,B
      READ(5,FMT=10,END=1)A,B
10    FORMAT(2F10.4))
      NOMORE=.FALSE.
      RETURN
1     NOMORE=.TRUE.
      END
```

However it is good practice to explicitly type all variables to avoid potential errors.

Summary

In addition to reals and integers, Fortran recognises two other types of numerical data — double precision and complex.

- DOUBLE PRECISION doubles the number of bits which a real number uses, thus extending the precision. The actual precision obtainable depends on the characteristics of the machine being used.
- COMPLEX is used to store and manipulate complex numbers those with a real and imaginary part.
- There are standard functions which allow conversion between the numerical data types — DBLE, CMPLX, REAL and INT.

Another type of data — logical — is also recognised. A LOGICAL variable may take one of two values — *true* or *false.*

- There are special operators for manipulating logicals .NOT., .AND. and .OR..
- Logical operators have a lower order of precedence than any others.

Any user defined functions which return these data types through the function name must provide the type explicitly on the FUNCTION statement, e.g.

```
LOGICAL FUNCTION FLAG(A)
DOUBLE PRECISION FUNCTION ADD(X,Y)
COMPLEX FUNCTION CSINH(Z)
```

Problems

1. Why are the full stops needed in a statement like A = .TRUE.?

2. Generate a truth table like the one given in this chapter.

3. Write a logical function, which will read in numerical data from the terminal, but will ‘flag’ any data which is negative, and will also turn these negative values into positive ones. Find the largest value as well.

4. Write a program to read in an arbitrary number of numbers, using a function called MORDAT to detect if there is more data.

5. The program used in chapter 12 which calculated the roots of a quadratic had to abandon the calculation if the roots were complex. You should now be able to remedy this, remembering that it is necessary to declare any complex variables. Instead of raising the expression to the power 0.5 in order to square root it, use the function SQRT. If you manage this to your satisfaction, try your skills on the roots of a cubic (see the problems in chapter 12).

17

Characters

These metaphysics of magicians,
And necromantic books are heavenly;
Lines, circles, letters and characters;

Christopher Marlowe, 'The Tragical History of Doctor Faustus'

Aims

The aims of this chapter are:–

- to extend the ideas about characters introduced in earlier chapters
- to demonstrate that this enables us to solve a whole new range of problems in a satisfactory way

Introduction

Character information is of a fundamentally different type to numerical information. There are no concepts of arithmetic associated with the manipulation of characters. The character variable is strongly typed.

The basic unit is an individual character — any character which is available on your keyboard. We will restrict ourselves to the Fortran character set, that is:

the **alphanumerics**: A–Z and 0–9

the **operators**: + – / *

the **punctuation**: , . : space ( ) = ’ $

Any of these can be a character. Because the Fortran standard wished to make character manipulation and processing general across a wide range of machines, it was necessary to define a special variable type, CHARACTER. One important consequence of this is that it is not possible to store information other than character information in this variable type (no more than you would expect to hold a real number in a logical variable). You will remember that it is syntactically correct to say, e.g.

```
A=B
```

where A and B are real, complex, double precision or integer, and no matter what combination there is, something will happen (or rather, will be allowed to happen). However, if either of A or B is a character variable, the other must be too. Remember that the integer 2 and the character 2 are not the same as far as Fortran is concerned.

We may declare our character variables

```
CHARACTER A, STRING, LINE
```

Notice that there is no default typing of the character variable, and we can use any convenient name within the normal Fortran conventions. In the declaration above, each character variable would have been permitted to store one character. This is limiting, and, to allow character strings which are several units long, we have to add a piece of information

```
CHARACTER A*10, STRING*16, LINE*80
```

This indicates that A holds 10 characters, STRING holds 16, and LINE holds 80. If all the character variables in a single declaration contain the same number of characters, we may abbreviate the declaration to

```
CHARACTER*80 LIST, STRING, LINE
```

But we cannot mix both forms in the one declaration. We can now assign data to these variables, as follows:–

```
      A='FIRST ONE '
      STRING='A LONGER ONE    '
      LINE='THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG'
```

The delimiter apostrophe (') is needed to indicate that this is a character string (otherwise the assignments would have looked like invalid variable names).

This instantly raises the problem 'How do I get an apostrophe into my character string?' The way chosen in Fortran is to represent a single apostrophe within a character string by two consecutive apostrophes, e.g.

```
      STRING='ARTHUR''S PROGRAM'
```

Note that we do not use ", the double quote, which is not part of the defined character set anyway. Since each pair of apostrophes within a character string counts only as a single character, we can have a situation like:

```
      CHARACTER PRIME*1
      PRIME=''''
```

Although it looks rather quaint, this is quite straightforward, and, under the right circumstances might be quite useful. The first and last apostrophes are the delimiters of the string, and so contribute 'nothing' to the string itself. The pair of apostrophes is considered to be a single character which will be stored in a character variable PRIME.

We can read and write character variables through the A format.

```
      CHARACTER STRING*16, A*10
      READ (UNIT=5,FMT='(A)') STRING
```

is equivalent to

```
      READ(UNIT=5,FMT=100) STRING
100   FORMAT(A)
```

and also to

```
      CHARACTER FORM*3
      FORM='(A)'
      .
      READ(UNIT=5,FMT=FORM)STRING
```

Note that, in using the first form, it is necessary to 'delimit' the formal elements of the format in primes to say, in effect, that this is a character string. The READ statements would allow us to read in a character variable of 16 characters from logical unit 5. Similarly

```
      WRITE(UNIT=6,FMT='(A)') STRING
```

or

```
      WRITE(UNIT=6,FMT=101) STRING
101   FORMAT(A)
```

would allow us to write this string out. Note that a statement like

```
      READ(UNIT=5,FMT=10) A,STRING
10   FORMAT(2A)
```

would read A of length 10, and STRING of length 16, although the format itself only indicates that two character variables are being read, and has nothing to say about their lengths. The length information is implicit to the variables themselves. The lengths are fixed in the declaration of the character variables. This seems to indicate that we are restricted in what we can do with characters, since there seems to be some limitation on sizes. This is not true however. There are manipulations we can perform on character strings which makes it a very flexible variable type indeed.

The first manipulator is a new operator — the concatenation operator //. With this operator we can join two character variables to form a third, as in

```
      CHARACTER FIRST*5, SECOND*5, THIRD*10
      FIRST='THREE'
      SECOND='BLIND'
      .
      .
      THIRD=FIRST//SECOND
      .
      THIRD=FIRST//'MICE'
```

Where there is a discrepancy between the created length of the concatenated string and the declared lengths of the character strings, truncation will occur. For example

```
      THIRD=FIRST//' BLIND MICE'
```

will only append the first five characters of the string ‘ BLIND MICE’ – that is ‘ BLIN’, and THIRD will therefore contain ‘THREE BLIN’.

What would happen if we assigned a character variable of length ‘n’ to a string which was shorter than n? e.g.

```
      CHARACTER C2*4
      C2='AB'
```

The remaining two characters are considered to be blank, that is, it is equivalent to saying

```
      C2='AB  '
```

However, while the strings ‘AB’ and ‘AB   ’ are equivalent, ‘AB’ and ‘   AB’ are not. In the jargon, the character strings are always *left justified,* and the *unset* characters are trailing blanks.

If we concatenate strings which have ‘trailing blanks’, the blanks, or spaces, are considered to be legitimate characters, and the concatenation begins after the end of the first string. Thus

```
CHARACTER*4 C2,C3
CHARACTER JJ*8
C2='A'
C3='MAN'
JJ=C2//C3
PRINT*, 'THE CONCATENATION OF ',C2,' AND ',C3,' IS'
PRINT*,JJ
```

would appear as

```
THE CONCATENATION OF A    AND  MAN GIVES
A   MAN
```

at the terminal.

Sometimes we need to be able to extract parts of character variables — sub-strings. The actual notation for doing this is a little strange at first, but it is very powerful. To extract a sub-string we must reference two items;

- (i) the position in the string at which the sub-string begins,

and

- (ii) the position at which it ends.

e.g.

```
STRING='SHARE AND ENJOY'
```

We may extract parts of this string

```
BIT=STRING(3:5)
```

would place the characters ‘ARE’ into the variable BIT. This may be manipulated further

```
BIT1=STRING(2:4)//STRING(9:9)
BIT2=STRING(5:5)//STRING(3:3)//STRING(1:1)//STRING(15:15)
```

Note that to extract a *single* character we reference its beginning position and its end (i.e. repeat the same position), so that

```
STRING(3:3)
```

gives the single character ‘A’. The sub-string reference can cut out either one of the two numerical arguments. If the first is omitted, the characters up to and including the reference are selected, so that

```
        SUB=STRING(:5)
```

would result in SUB containing the characters ‘SHARE’. When the second argument is omitted, the characters from the reference are selected, so that

```
        SUB=STRING(11:)
```

would place the characters ‘ENJOY’ in the variable SUB. In these examples it would also be necessary to declare STRING, SUB, BIT, BIT1 and BIT2 as CHARACTER type, of some appropriate length.

Character variables may also form arrays.

```
        CHARACTER*10 A
        DIMENSION A(20)
```

sets up a character array of twenty elements, where each element contains ten characters. In order to extract sub-strings from these array elements, we need to know where the array reference and the sub-string reference are placed. The array reference comes first, so that

```
        DO 1 I=1,20
            FIRST=A(I)(1:1)
1       CONTINUE
```

places the first character of each element of the array into the variable FIRST. The syntax is therefore ‘position in array, followed by position within string’.

Any argument can be replaced by a variable:

```
        STRING(I:J)
```

This offers interesting possibilities, since we can, for example, strip out blanks from a string

```
        CHARACTER*80 STRING, STRIP
        INTEGER IPOS,I,LEN
        IPOS=0
        DO 1 I=1,LEN
            IF(STRING(I:I).NE.' ') THEN
                IPOS=IPOS+1
                STRIP(IPOS:IPOS)=STRING(I:I)
            ENDIF
1       CONTINUE
        PRINT*,STRING
        PRINT*,STRIP
```

Character functions

There are special functions available for use with character variables: INDEX will give the starting position of a string within another string. If, for example we were looking for all occurrences of the string ‘GEOLOGY’ in a file, we could construct something like

```
        CHARACTER L*80
        INTEGER I
        .
1       READ (*,END=10,FMT='(A)') L
            I=INDEX(L,'GEOLOGY')
            .
        GO TO 1
10      CONTINUE
```

There are two things to note with this function. Firstly, it will only report the first occurrence of the string in the line; any later occurrences in any particular line will go unnoticed, unless you account for this in some way. Secondly, if the string does not occur, the result is zero.

LEN provides the length of a character string. This function is not immediately useful, since you really ought to know how many characters there are in the string. However, as later examples will show, there are some cases where it can be useful. Remember that trailing blanks do count as part of the character string, and contribute to the length.

The next group of functions need to be considered together. They revolve around the concept of a collating sequence. In other words, each character used in Fortran is ordered as a list, and given a corresponding ‘weight’. No two weights are equal. Although Fortran has only 49 defined characters, the machine you use will generally have more; 95 printing characters a typical minimum number. On this type of machine the weights would vary from zero to 94. There is a defined collating sequence, the ASCII sequence, which is likely to be the default. The parts of the collating sequence which are of most interest are fairly standard throughout all collating sequences.

In general, we are interested in the numerals (0–9), the alphabetics (A–Z) and a few odds and ends like the arithmetic operators (+ – / *), some punctuation (. and ,) and perhaps the prime. As you might expect, 0–9 carry successively higher weights (though not the weights 0 to 9), as do A to Z. The other odds and ends are a little more problematic, but we can find out the weights through the function ICHAR. This function takes a single character as argument, and returns an integer value. The ASCII weights for the alphanumerics are as follows:–

0–9	48–57
A–Z	65–90

One of the exercises is to determine the weights for other characters. The reverse of this procedure is to determine the character from its weighting, which can be achieved through the function CHAR. CHAR takes an integer argument and returns a single character. Using the ASCII collating sequence, the alphabet would be generated from

```
      DO 1 I=65,90
         PRINT*,CHAR(I)
1     CONTINUE
```

This idea of a weighting may then be used in four other functions:–

Function	Action
LLE	lexically less than or equal to
LGE	lexically greater than or equal to
LGT	lexically greater than
LLT	lexically less than

In the sequence we have seen before, A is lexically less than B; i.e. its weight is less. Clearly, we can use ICHAR and get the same result. For example

```
      IF(LGT('A','B')) THEN
```

is equivalent to

```
      IF(ICHAR('A').GT.ICHAR('B')) THEN
```

but these functions can take character string arguments of any length. They are not restricted to single characters. Although the functions look a little like the logical operators, they must be used as functions, in the manner shown.

These functions provide very powerful tools for the manipulation of characters, and open up wide areas of non-numerical computing through Fortran. Lots of text formatting and word processing applications may now be tackled (conveniently ignoring the fact that lower case characters may not be available).

Remember that any functions you write which return character results must be explicitly declared as character type on the function statement, e.g.

```
      CHARACTER FUNCTION OMEGA(A,B)
      CHARACTER OMEGA*10, A*5, B*5
      .
      OMEGA=A//B
      .
      END
```

Just to show how you might wish to use a character variable which is given a *variable* length, the previous example might be re-written

```
      CHARACTER FUNCTION OMEGA(A,B)
      CHARACTER*10 OMEGA
      CHARACTER *(*) A,B
      INTEGER LA,LB
      .
      .
      LA=LEN(A)
      LB=LEN(B)
      IF(LA+LB.LE.10) THEN
            OMEGA=A//B
      ELSE
            OMEGA='TOO LONG'
      ENDIF
      .
      .
      END
```

The statement

```
      CHARACTER *(*) A,B
```

indicates that we do not know the lengths of A and B, although they will have been set in the calling routine(s). Note the strange syntax here, where the second asterisk must be contained within brackets. The function also uses the LEN function, just to filter out those occasions where the combined length of the strings A and B is greater than 10.

Example

One convenient application of character variables is in creating simple graphs at the terminal. These can never be very accurate, but they can be quick and informative.

```
      PROGRAM DIAGRM
      REAL X,Y,XMIN,YMIN,XMAX,YMAX,XW,YW
      DIMENSION X(10),Y(10)
      INTEGER I,ILEN,JLEN,N, IPOS, JPOS
      CHARACTER*40 DIAG(20)
      PARAMETER (ILEN=20,JLEN=40)
      OPEN(UNIT=6,FILE='OUTPUT')
C  GETTING THE DATA IN
      WRITE(UNIT=6,FMT=104)
      READ *,N
      WRITE(UNIT=6,FMT=101) N
      READ *,(X(I),Y(I),I=1,N)
      WRITE(UNIT=6,FMT=102)
      READ *,XMIN,XMAX
```

```
        WRITE(UNIT=6,FMT=103)
        READ *,YMIN,YMAX
C  CALCULATING SCALING CONSTANTS
        XW = (XMAX-XMIN)/(JLEN-1)
        YW = (YMAX-YMIN)/(ILEN-1)
C INITIALISE THE CHARACTER STRING TO ALL BLANKS
        DO 1 I=1,ILEN
            DIAG(I)=' '
1       CONTINUE
        DO 2 I=1,N
            JPOS=(X(I)-XMIN)/XW+1
            IPOS=(Y(I)-YMIN)/YW+1
C           ELIMINATING POINTS OUTSIDE THE DIAGRAM
            IF(IPOS.LT.1.OR.IPOS.GT.ILEN)THEN
                WRITE(UNIT=6,FMT=100) X(I),Y(I)
            ELSEIF(JPOS.LT.1.OR.JPOS.GT.JLEN)THEN
                WRITE(UNIT=6,FMT=100) X(I),Y(I)
            ELSE
C               THESE ARE INSIDE
                DIAG(21-IPOS)(JPOS:JPOS)='*'
            ENDIF
2       CONTINUE
C     NOW WRITE OUT THE COMPLETED DIAGRAM
        DO 3 I=1,ILEN
            WRITE(UNIT=6,FMT='(1X,'':'',A)') DIAG(I)
3       CONTINUE
        WRITE(UNIT=6,FMT='(1X,40(''-''))')
100     FORMAT(' POINT OUT OF RANGE ',2F10.4)
101     FORMAT(' GIVE ',I5,' PAIRS OF POINTS,X-VALUE,Y-VALUE')
102     FORMAT(' GIVE MAXIMUM AND MINIMUM FOR X-VALUES')
103     FORMAT(' GIVE MAXIMUM AND MINIMUM FOR Y-VALUES')
104     FORMAT(' GIVE NUMBER OF PAIRS FOR PLOTTING')
        END
```

One of the points to note in this program is the way in which we make sure that all the graph points lie within the plotting area. Trying to address points outside this area can pose problems. Note also that an x-axis and a y-axis are printed on the plot.

There are many problems that require the use of character variables. These range from the ability to provide simple titles on reports, or graphical output, to the provision of a natural language interface to one of your programs, i.e. the provision of an English-like command language. *Software Tools*, Kernighan and Plauger contains many interesting uses of characters in Fortran.

Summary

• Characters represent a different data type to any other in Fortran, and as a consequence there is a restricted range of operations which may be carried out on them.

• A character variable has a length which must be assigned in a CHARACTER declaration statement.

• Character strings are delimited by apostrophes. Within a character string, the blank is a significant character.

• Character strings may be joined together (concatenated) with the // operator.

• Sub-strings, occurring within character strings, may be also be manipulated. There are a number of functions especially for use with characters — INDEX, LEN, CHAR, ICHAR, LLE, LGE, LGT and LLT.

Problems

1. Suggest some circumstances where PRIME='''' might be useful.

2. Write a program to write out the weights for the Fortran character set.

3. Use the INDEX function in order to find the location of all the strings 'IS' in the following data;

IF A PROGRAMMER IS FOUND TO BE INDISPENSABLE, THE BEST THING TO DO IS TO GET RID OF HIM AS QUICKLY AS POSSIBLE.

4. Find the 'middle' character in the following strings. Do you include blanks as characters? What about punctuation?

PRACTICE IS THE BEST OF ALL INSTRUCTORS. EXPERIENCE IS A DEAR TEACHER, BUT FOOLS WILL LEARN AT NO OTHER.

5. In English, the order of occurrence of the letters, from most frequent to least is:–

E, T, A, O, N, R, I, S, H, D, L, F, C, M, U, G, Y, P, W, B, V, K, X, J, Q, Z.

Use this information to examine the two files given in appendix B (one is a translation of the other) to see if this is true for these two extracts of text. The second text is in medieval Latin (c. 1320). Note that a fair amount of compression has been achieved by expressing the passage in Latin rather than modern English. Does this provide a possible model for information compression?

6. A very common cypher is the substitution cypher, where, for example, every letter A is replaced by (say) an M, every B is replaced by (say) a Y, and so on. These encyphered messages can be broken by reference to the frequency of occurrence of the letters (given in the previous question). Since we know that (in English) E is the most commonly occurring letter, we can assume that the

most commonly occurring letter in the encyphered message represents an E; we then repeat the process for the next most common and so on. Of course, these correspondences may not be exact, since the message may not be long enough to develop the frequencies fully. However, it may provide sufficient information to break the cypher. The file given in Appendix C contains an encoded message. Break it. Clue — *‘Pg Fybdujuvef jo Tdjfodf’*,Jorge Luis Borges.

7. The simple graph plotting program given in the chapter could be improved by adding titles, by making the calculation of minima and maxima automatic, and perhaps by identifying places where two points fall on the same plotting location. Try to implement some of these enhancements.

18

Subroutines

A man should keep his little attic stacked with all the furniture he is likely to use, and put the rest away in the lumber room of his library, where he can get it if he wants.

Sir Arthur Conan Doyle, Five Orange Pips.

Aims

The aims of this chapter are:–

- to introduce another way of breaking problems down into small self-contained pieces
- to illustrate the use of subroutines
- to introduce the idea of a *library* of subroutines
- to make you aware of the expertise that you can draw on and the time you can save by the use of these subroutine libraries

Introduction

You have already seen how one can use functions to help break a problem down into manageable pieces. Fortran provides another more general way of doing this using a SUBROUTINE.

The structure is slightly different from that of a function,

```
SUBROUTINE MULT(X,Y,Z,FUN)
REAL X,Y,Z,FUN
FUN=X*Y**Z
END
```

and the reference is also slightly different:–

```
PROGRAM SIMPLE
REAL A2,A,B,C,FN,X
.
CALL MULT(A,B,C,FN)
.
A2=FN/X
.
END
```

Notes

1. The type of MULT has no effect here.

2. The order of the arguments is again significant.

3. The names used in the calling routine have no effect in the sub-program. A variable A in the calling routine has no relationship whatsoever with a variable A in the subroutine or function.

4. We have to introduce a new variable (FN) to hold the result.

5. We must not name any variables MULT. This will cause great confusion.

6. To use the subroutine we CALL it.

The fourth condition (introduction of the variable FN) is not restrictive, since we could actually write

```
PROGRAM SIMPLE
REAL A2,A,B,C,X
.
.
CALL MULT(A,B,C)
.
A2=C/X
.
END
```

```
      SUBROUTINE MULT(X,Y,Z)
      REAL X,Y,Z
      Z=X*Y**Z
      END
```

While functions must have at least one argument, subroutines can have any number, including none at all. We might ask whether such a routine would be of any value. If you return to examine some of the intrinsic functions, you may recall that some of them could take variable numbers of arguments. You cannot write such functions in standard Fortran.

Making subroutines (and functions) more general

Lets return to the functions for finding the minimum and maximum; these two functions could be combined into a single subroutine, such as

```
      SUBROUTINE MINMAX(V,N,VMAX,VMIN)
      REAL V,VMIN,VMAX
      INTEGER I,N
      DIMENSION V(100)
      VMIN=V(1)
      VMAX=VMIN
      DO 1 I=2,N
         IF(V(I).GT.VMAX) THEN
            VMAX=V(I)
         ELSEIF(V(I).LT.VMIN) THEN
            VMIN=V(I)
         ENDIF
1     CONTINUE
      END
```

To use this sub-program, we use the statement

```
      CALL MINMAX(X,N,XMAX,XMIN)
```

where XMAX and XMIN are the results, and the other arguments are as before. When you use subroutines more extensively, you will begin to discover that it is irritating to have to dimension arrays to a fixed amount in the subroutine, as in

```
      REAL X,SUM
      DIMENSION X(100)
      .
      CALL ADD(X,SUM)
      .
      END
      SUBROUTINE ADD(A,TOTAL)
      REAL A,TOTAL
      DIMENSION A(100)
      INTEGER I
```

```
        TOTAL=0.0
        DO 1 I=1,100
            TOTAL=TOTAL+A(I)
1       CONTINUE
        END
```

The argument X is an array dimensioned to 100 in the calling routine, and is again dimensioned to 100 in the subroutine. This is not very general, and there are ways in which the subroutine can be made more flexible. It is possible to dimension the array to an arbitrary length N, as long as N does not exceed the size of the equivalent array in the calling routine, e.g.

```
        REAL X,SUM
        DIMENSION X(100)
        CALL ADD(X,SUM,100)
        .
        SUBROUTINE ADD(A,TOTAL,N)
        REAL A,TOTAL
        INTEGER N
        DIMENSION A(N)
        TOTAL=0.0
        DO 1 I=1,N
        .
```

We have acquired an extra argument, but increased the scope of the routine greatly, since we may now use it in other situations, such as

```
        DIMENSION X(100),Y(10),Z(50)
        .
        .
        CALL ADD(X,XSUM,100)
        .
        .
        CALL ADD(Y,YSUM,10)
        .
        .
        CALL ADD(Z,ZSUM,50)
        .
        .
        etc
```

The routine we began with could not have handled this ‘adjustable’ array size, and would have plodded away to 100 each time. Although we discussed sub-programs earlier in terms of avoiding duplication, you can see that the similarity between the sequence of events need not be exact. The problem started out as ‘add 100 numbers together’, but ended up as ‘add N numbers together’. Of course, the argument on the CALL which determines the array length in the subroutine need not be a number, but could be a variable, as in:–

```
      NX=100
      NY=10
      CALL ADD(X,XSUM,NX)
      CALL ADD(Y,YSUM,NY)
```

It is important to notice that we may only use this technique when space has already been allocated for the arrays. A structure like

```
      SUBROUTINE ADD2(X,Y,N)
      REAL X,Y,Z
      INTEGER I,N
      DIMENSION X(N),Y(N),Z(N)
      DO 1 I=1,N
         Z(I)=X(I)+Y(I)
1     CONTINUE
```

is not permitted, since Z is not an argument of the subroutine and has had no dimension associated with it to set aside space for its contents. The setting aside of space is accomplished at compile time, while this routine would expect Z to be set up at run time ('dynamically'). Essentially, Fortran is a static language, which sets aside all space required at the stage before running.

Similarly, you cannot increase the size of an array beyond the limit it was originally given in the DIMENSION statement.

It is often the case that we wish to manipulate arrays and for example, join them together to make larger entities. We might have two vectors, A (of length 100) and B (of length 50), which we wish to join together to form C (of length 150). If we generalise this to make A of length M and B of length N, the length of C will become M+N. Unfortunately, Fortran does not permit a DIMENSION statement to have an argument like M+N in a sub-program. It does however permit the use of the asterisk to take care of situations like this.

```
      SUBROUTINE JOIN(A,M,B,N,C)
      REAL A,B,C
      INTEGER I,M,N
      DIMENSION A(M),B(N),C(*)
      DO 1 I=1,M
         C(I)=A(I)
1     CONTINUE
      DO 2 I=1,N
          C(I+M)=B(I)
2     CONTINUE
      END
```

Again, C must have been dimensioned large enough in the calling routine.

One of the problems of using subroutines with variable length arrays stems from the way in which Fortran 'holds' or stores arrays in memory. The array is not held as a two (three, ... n) dimensional structure, but as a vector. This implies that a general structure like

```
DIMENSION X(20,20)
.
CALL SUB1(X,10)
CALL SUB1(X,15)
CALL SUB1(X,16)
.
END
SUBROUTINE SUB1(A,N)
DIMENSION A(N,N)
```

will not work. The array carried into the subroutine will not be a ten by ten matrix, then a fifteen by fifteen, and so on. The array will be stored as a single vector, taken column-wise from the twenty by twenty array. To get around this problem, we must take the ‘true’ dimension of the array into the subroutine:

```
.
CALL SUB1(X,20,10)
.
END
SUBROUTINE SUB1(A,LENGTH,N)
DIMENSION A(LENGTH,LENGTH)
```

An alternative mechanism, which avoids using some of the extra arguments, is through the use of the asterisk. The asterisk may be used in place of the *last* dimension of the array:

```
DIMENSION B(20,10)
N=20
.
CALL SUB2(B,N)
END
SUBROUTINE SUB2(C,M)
DIMENSION A(M,*)
```

This will really only be useful for ‘rectangular’ arrays, i.e. where the maximum dimension bounds are not the same.

The asterisk also becomes more valuable in its use with characters. When it is necessary to take character strings into subroutines, we may have situations where we do not know the string lengths. Of course, we could have used the LEN function to find out, but we may also use something like

```
SUBROUTINE STRING(X)
CHARACTER*(*) X
PRINT*,X
```

Note the use of the brackets around the asterisk, which indicates the adjustable bound.

Subroutines may call other subroutines, but not recursively. If you don't know what recursion is, you won't even notice.

Example

Solving sets of simultaneous equations can be programmed fairly readily, especially using a technique known as Gaussian elimination; essentially this is the same method that you would use (perhaps almost intuitively) to solve for one term, and then 'back substitute'.

```
      SUBROUTINE SOLVE(A,B,N,X)
      INTEGER J,K,L,N
      REAL A,B,X
      DIMENSION A(N,N),B(N),X(N)
C
C SOLVES A SET OF N SIMULTANEOUS EQUATIONS, OF THE FORM
C A(1,1)*X(1) + A(1,2)*X(2) + A(1,3)*X(3) .... = B(1)
C A(2,1)*X(1) + A(2,2)*X(2) + A(2,3)*X(3) .... = B(2)
C
C A(N,1)*X(1) + A(N,2)*X(2) + A(N,3)*X(3) .... = B(N)
C
C INPUT
C THE MATRIX A CONTAINS THE COEFFICIENTS ON THE LHS
C THE VECTOR B CONTAINS THE VALUES ON THE RHS
C OUTPUT
C THE VECTOR X RETURNS THE VALUES AS ABOVE
C NOE THAT THE CONTENTS OF A AND B ARE CHANGED
C
      DO 1 K=1,N
        DO 2 J=K+1,N
          Y=-A(J,K)/A(K,K)
          DO 3 L=K,N
            A(J,L)=A(J,L)+Y*A(K,L)
3         CONTINUE
          B(J)=B(J)+Y*B(K)
2       CONTINUE
1     CONTINUE
C START THE BACK SUBSTITUTION
      X(N)=B(N)/A(N,N)
      DO 4 J=N-1,1,-1
        Y=B(J)
        DO 5 K=J+1,N
          Y=Y-A(J,K)*X(K)
5       CONTINUE
        X(J)=Y/A(J,J)
4     CONTINUE
      END
```

It must be noted that although this method is easy to program it can be numerically unstable in the presence of rounding errors. Gaussian elimination should therefore only be performed with pivots. Ralston and Rabinowitz provides further information for the interested reader.

Available subroutine libraries

One reason we suggested for using sub-programs was to avoid duplication. Very often, parts of the problems we wish to tackle have already been solved by others. There seems little point in re- inventing the wheel if we can use the accumulated knowledge and expertise of other wheel-wrights.

There are a number of ‘libraries’ of routines available which will allow access to this pool of expertise.

One notable library, available in all British Universities, is the NAG, or Numerical Algorithms Group, library. This is a collection of literally hundreds of routines (some are functions, some are subroutines), all of which may be called from a Fortran program. The general section headings are given Appendix D, and indicate the areas which are currently available. The NAG library contains a great many routines, arranged in some kind of useful way. A good problem to solve, to prove you can program, is arranging real numbers in order, either ascending or descending. It is not too difficult to do this, but doing it efficiently, either in terms of time or storage, is a problem which has taxed many minds, and the NAG library contains a variety of routines to do this well e.g. routine M01CAF, which may be used in the following way:

```
PROGRAM NAGEX1
REAL A
INTEGER IFAIL,N
DIMENSION  A(10)
N=10
IFAIL=0
READ *,A
CALL M01CAF(A,N,’DESCENDING’,IFAIL))
PRINT *,A
END
```

This takes a vector A of length 10, and sorts it into descending order. Within a program, use of a NAG routine is the same as any other subroutine or function.

Note that the name of the sub-program is a little strange, and is not the friendly mnemonic which we might expect. However, there is a good reason for this. The first two letters indicate the chapter in the manual; the next two digits are a reference within the chapter; the next two letters identify the particular sub-program, while the last letter (F) indicates the language (Fortran). The form of the name is close to the standard naming procedure used by the Association for Computing Machinery (ACM), CERN (European Organisation for Nuclear Research) etc., and is termed the ‘modified SHARE index’. When using subroutine libraries, it is likely that you will have to allocate some work-space for the routines. In this example the algorithm used is very fast and requires no extra space, though for most routines you do. The size of these arrays is determined by the size of the other arrays which actually store the data and results.

Most subroutine libraries have an error parameter of some description. We will discuss the one that NAG provide to give a concrete idea of the use of a parameter like this. It is called the IFAIL parameter. For most routines the IFAIL parameter has two purposes:–

- (i) to allow you to specify the action to be taken if an error is detected

and

- (ii) to inform you of the success or failure of the routine.

Thus for (i), you must assign a value to IFAIL before entering the routine. Note that IFAIL is reset by the routine, so you cannot pass a value, but must use a variable name, which has previously been given a value. You may set either a ‘hard fail option’, or a ‘soft fail option’. Hard fail (setting IFAIL to zero) instructs the program to terminate if an error is detected and an appropriate error message is printed together with the value of IFAIL; soft fail (IFAIL=1) instructs the routine to ‘recover’, and return to the calling routine – the value given to IFAIL on returning will reflect the nature of the error. If you select the soft fail option, a successful call would be represented by an IFAIL value of zero. If you use the soft fail, you must test the value of IFAIL, or an error may go undetected, and your subsequent results may be suspect. This is not needed if the ‘soft fail’, (IFAIL=–1) option is taken because an error message will be printed before recovering.

NAG have a policy of continuous improvement of their algorithms, and make changes from version to version. In theory, each new version will contain the elements of the previous ones. Extensive documentation is available for this library.

Plotting

Another important area which is usually addressed through libraries is the extension of Fortran to output devices like graphics terminals, Tektronix display units, or pen plotters. You probably have access to some sort of plotting device. These may be controlled through Fortran programs which use subroutine libraries like UNIRAS, NAG Graphics or something similar. We will only describe a very simple conceptual plotter here.

First, assume that there is a notion of position; the pen (or light beam, or whatever is creating the ‘vector’) has a position. Secondly, when the pen is moved, we may either move with the pen down (when a line would be drawn), or with the pen up (when no line would be made); at the end of a movement, the pen has a new position. The next movement will be made from this position to the next 'new' one. A possible subroutine call would be:

```
CALL PLOT(X1,Y1,LINE)
```

where X1 and Y1 are the cartesian co-ordinates of the 'next' point — the one to which the pen will move. By definition the pen already has a position from which to move. The variable LINE specifies whether a line is being drawn or not (perhaps a logical or integer variable). With this power to move, to create lines or not, very be very many more subroutines than this movement primitive in a plotting library. They will allow you to write text (for titling information), write numbers (for axis scaling), draw all sorts of symbols (for graphs), and much more.

Algorithm libraries

Another way in which you can draw on the expertise of others is by the use of published algorithms. One of the best known is the one published by the Association for Computing Machinery and often called TOMS (for Transactions on Mathematical Software). This library may well be available in machine readable form at your site.

Summary

• One key way of making programs more modular is through the use of subroutines, where a single task, or group of related tasks may be tackled.

• Subroutines are invoked differently from functions. They are CALLed; however the same notions of communication through the argument list applies. The name of the subroutine does not return a value, and is used merely as a convenient mnemonic.

• Subroutines using arrays may be made more general by permitting the arrays to take variable length bounds; this applies only to arrays which appear in the argument list. Arrays may be dimensioned to an integer variable, which is also an argument, or to an asterisk. Only the last array bound may be given the value asterisk.

• There is no restriction on the number of arguments to a subroutine, and the apparent type of the name has no relevance.

• Subroutine libraries are widely available and permit use of the expertise of others. Such libraries also allow access to more specialised input and output devices.

• There are also many published algorithms which provide a basis for specialised subroutines and functions.

Problems

1. Find the eigen-values of the following matrix. You don’t even have to know what eigen-values are to do this one.

$$\begin{array}{cccc} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{array}$$

2. Write a sub-program to concatenate the character vector A and the character vector B, leaving a space between them. Place the result in C. Is it general, and if not, what are the restrictions?

3. The subroutine SOLVE contains some defects; it will only solve N by N sets of equations, where N is the same value in the calling routine. Can you correct this? Test the subroutine on the following examples:

$$\begin{aligned} 2x + 3y + 4z &= 11 \\ 6x + 5y + 10z &= 43 \\ 5x - 11y + 8z &= 20 \end{aligned}$$

and

$$\begin{aligned} 2x + 5y + 3z - w &= 0 \\ 6x - y + z + 3w &= 4 \\ 4x + y - 2z + 3w &= 23 \\ 0x + 7y + 0z - w &= 19 \end{aligned}$$

This last example reveals another inadequacy of the subroutine. This may provide a good chance to test out any facilities like ‘post-mortem dump’. Can you solve the problem, or failing that, provide an error detection mechanism?

4. Many of the programs presented in this book could be written as subroutines. The plotting program in the previous chapter could make a useful subroutine. Turn it into a subroutine, giving attention to the dimensions of the finished diagram. Large diagrams are fine on line-printers, but awkward on small *vdu* screens, or over very slow lines (30 characters per second). Take these considerations into account.

19

Files

It is a capital mistake to theorise before one has data.

Sir Arthur Conan Doyle.

Aims

The aims of this chapter are:–

- to review the process of file creation at a terminal
- to introduce more formally the idea of the file as a fundamental entity
- to show how files can be declared explicitly by the OPEN and CLOSE statements
- to introduce the arguments for the OPEN and CLOSE statements
- to demonstrate the interaction between the READ/WRITE statements and the OPEN/CLOSE statements

Introduction

While you are working interactively, on a terminal, you will be working with files; files that contain programs, files that contain data, and perhaps files that are libraries. The file is fundamental to most modern timesharing operating systems, and almost all operations are carried out on files.

In this chapter we are going to extend some of your ideas about files. Let us consider what kinds of files you have met so far:–

1) Text files. These are the source of your programs, documents, reports etc. They can be examined by printing them. They can also be transmitted round a computer system fairly easily. A file sent to a printer is a text file.

2) Data files. These are generally a variation on 1. They can be printed in much the same way as a text file.

3) Binary, object or relocatable files. Typically these will be the output from a compiler. They cannot be printed. To examine files like these you need to use special utilities, provided by most operating systems.

The above categories account for the majority of files that you have met so far.

Let us now consider how we can manipulate files using Fortran. They will generally be data files, and will thus be text files. They can therefore be listed etc., using standard operating system commands.

Files in Fortran

These allow us to associate a logical unit number with any arbitrary file name during the running of the program, e.g.

```
OPEN(UNIT=1,FILE='DATA')
```

would associate the name DATA and the logical unit 1, so that

```
READ(UNIT=1,FMT=100) X
```

would read from DATA. Note that for this to work on some operating systems the file DATA must have been ‘local’ to the session; we specify the name as a character variable. If we then wanted to use a subsequent data file, we could have another OPEN statement, but if we want to use the same logical unit number, we must first CLOSE the file.

```
CLOSE(UNIT=1,FILE='DATA')
```

before we

```
OPEN(UNIT=1,FILE='DATA2')
```

In this way we can keep referring to logical unit 1, but change the file associated with it. This can be useful in interactive programs where we wish to analyse different sets of data, e.g.

```
        PROGRAM FLEX
        REAL X
        CHARACTER*7 WHICH
        OPEN(UNIT=5,FILE='INPUT')
1       WRITE(UNIT=6,FMT='('' DATA SET NAME, OR END'')')
            READ(UNIT=5,FMT='(A)') WHICH
            IF(WHICH.EQ.'END') STOP
            OPEN(UNIT=1,FILE=WHICH)
            READ(UNIT=1,FMT=100) X
            .
            .
            .
            CLOSE(UNIT=1,FILE=WHICH)
        GO TO 1
        END
```

This last example also introduces the STOP statement. We have already encountered RETURN in functions, where they provide the potential for alternate exits from the sub-program. In a main program, RETURN is obviously inappropriate, but we may still require termination of the process at locations other than the END. The STOP statement may be employed, since it has the effect of terminating the program (wherever it occurs). The STOP statement may be placed in sub-programs, where it will also terminate the program.

One useful feature of the OPEN statement is that there are other parameters. What would happen, for example, if the file was not there? To take care of this you can use the ERR= and STATUS= keywords.

```
        OPEN(UNIT=1,FILE='DATA',ERR=10,STATUS='OLD')
```

If an error occurs during the attempt to open the file, control will transfer to the statement labelled 10. It is not sufficient that we use only the ERR= keyword in order to trap the absence of a file. We must also use the STATUS parameter. STATUS can be equated to one of four values,

```
        STATUS='OLD'
        STATUS='NEW'
        STATUS='SCRATCH'
        STATUS='UNKNOWN'
```

If we use STATUS='OLD' and the file is not present, this will cause an error condition and control will pass to whichever label is equated to ERR; if we say STATUS='NEW', we are creating a new file and it should not matter whether a file of the same name is present; 'SCRATCH' does not concern us, while 'UNKNOWN' implies that, if a file of the correct name is present use it, if not create a ’NEW’ one. If you omit the STATUS= keyword altogether, the value ’UNKNOWN’ will be assumed.

```
        OPEN(UNIT=1,FILE=’DATA’,ERR=10,STATUS=’OLD’)
        READ(UNIT=1,FMT=100) X
        .
        .
        STOP
10      WRITE(UNIT=6,FMT=200) DATA
200     FORMAT(’ Error in opening file, ’,A)
        END
```

Although this ERR= is rather like a GOTO, it is much more restricted, in that it is only available when an error occurs.

Summary of options on OPEN

UNIT The unit number of the file to be opened.

IOSTAT Integer variable given the value zero if there are no errors.

ERR In the event of an error control is transferred to the statement with this label.

FILE Character expression specifying the file name.

STATUS Character expression specifying the file status. It can be one of*’OLD’*, *’NEW’*, *’SCRATCH’*or*’UNKNOWN’*.

ACCESS Character expression specifying whether the file is to be used in a sequential or random fashion. Valid values are*’SEQUENTIAL’*(the default), or *’RANDOM’*.

FORM Character expression specifying one of:–

‘*FORMATTED’* if the file is opened for formatted i/o.

’UNFORMATTED’ if the file is opened for unformatted i/o.

’BUFFERED’ if the file is opened for buffered i/o.

The default is formatted for sequential access files and unformatted for direct access files. If the file exists, FORM must be consistent with its present characteristics.

RECL Integer variable or constant specifying the record length for a direct access file. It is specified in characters for a formatted file, and words for an unformatted file.

BLANK Character expression having one of the following values:–

’NULL’ if blanks are to be ignored on reading. Note that a field of all blanks is treated as 0!

'ZERO' if blanks are to be treated as zeros.

Some of the terms on the OPEN statement will be strange. We have introduced terms like 'unformatted', 'direct access', 'sequential', 'random' and 'buffered' without explanation. These terms are included for completeness, and it is probable that you will never need to use the facilities they provide. All file handling described in this book concerns sequential formatted files (the default type).

Summary

- The file is a fundamental entity within the operating system.
- Files may be manipulated in Fortran by associating their name with a unit number. All subsequent communication within the program is through the unit number.
- When a file is opened there are a large number of equatable keywords which may be employed to establish its characteristics.
- The default file type used in Fortran is *sequential formatted*, but several other esoteric types may be used.

Problems

1. Write a program to write the first 500 integers to a file using formatted i/o. Put 10 values on a line, with a blank as the first character of the line, and 8 columns allowed for each integer, with two spaces between each integer field.

Now write a program to read this file into an array, and write the numbers in reverse order over the original data. i.e. the data file now contains the first 500 numbers in descending order.

Now modify the first program to add the next 500 integers to the same file, so that the file now comprises the first 500 numbers in descending order, and the next 500 numbers in ascending order.

2. To write and maintain a crude data base of student details, we might do the following; create separate files for each year — CLAS1, CLAS2, CLAS3, or COF84, COF85, COF86, and so on. In either case there is an unchanging prefix, CLAS or COF, and a variable suffix, which identifies membership within the overall group. On each of the files we may wish to record details like; name, date of birth, address, courses taken, etc. Such files will require updating as details change, or as errors are noted. Write (or sketch out) a program which would select and maintain such records, and which would allow corrected files to be printed out. While you might feel that the most appropriate tool for this job is an editor, you might find this too powerful a tool. An editor can leave files in a sorry state. Naturally, any program like this should be helpful (so called 'user friendly'). Is this sort of information sensitive enough to require security checks and passwords?

20

Common and data statements

If we do not find anything pleasant,
at least we shall find something new.

Voltaire, Candide

Aims

The aims of this chapter are:–

- to introduce another way of providing communication between program, subroutines and functions
- to show how this mechanism can be used to enforce structure on the program
- to introduce a convenient mechanism for initialising data

Introduction

The communication between sub-programs has been so far through the argument list, e.g.

```
FUNCTION EVALUE(X,Y)
```

or

```
SUBROUTINE SOLVE(A,B,N,C)
```

There is one other way of sharing data between sub-programs — through the COMMON block.

A COMMON block is declared together with any other declarative statements at the beginning of the sub-program, e.g.

```
PROGRAM EXAMPLE
REAL X
DIMENSION X(100)
COMMON X
.
END
SUBROUTINE PASS1
REAL Y
DIMENSION Y(100)
COMMON Y
.
.
END
```

This has the effect of allowing the data stored in array X in the main program segment to be used as array Y in the subroutine. This particular form of the COMMON is known as ‘blank’ COMMON, since the common block has not been given a name. We can assign names to common blocks, so that we can distinguish them easily, e.g.

```
COMMON /A/ X(100)
COMMON /XRAY/ Y(20),B(30),Z
COMMON /HEAT/ A(25)
```

and so on. We may also write a blank common block as:

```
COMMON // XARRAY(25)
```

The name of a common block is only restricted in the sense that it must have no more than 6 alphanumeric characters. There is no concept of typing with common block names, and you may have all sorts of data types in the one common block, with one exception! If any character variable or character array is included in a common block, then all the entities in the block must be of type character.

The communication is effected through the sequence of the variables: a section of memory is set aside, so that, for example:–

```
        COMMON /DATA1/A(50),B(50)
```

sets aside 100 locations to a common block called DATA1. When DATA1 is declared in a sub-program, those 100 locations are available:

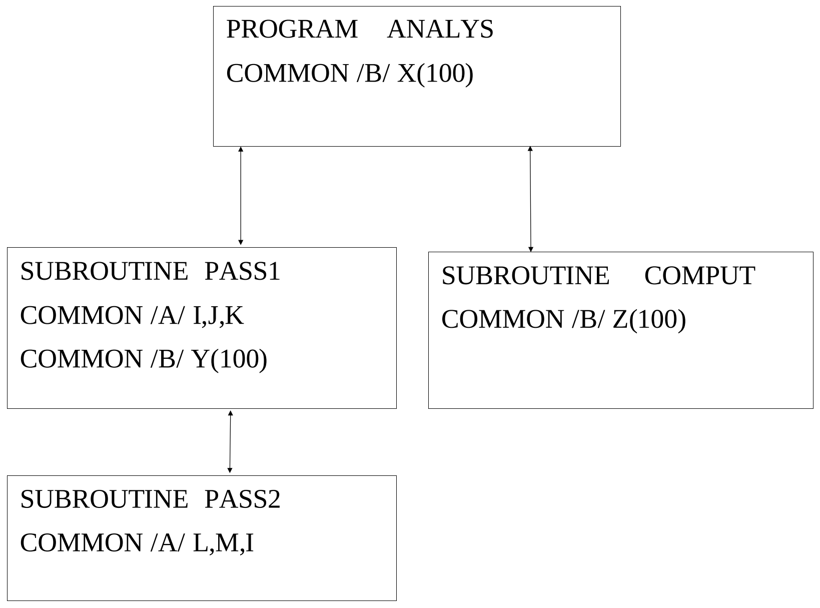
```
        COMMON /DATA1/X(10),Y(10),Z(80)
```

The first ten locations (from vector A in the calling routine), are usable in the array named X, the next ten in the array Y and so on. The fact that these had different descriptions in the calling routine is of no significance to Fortran.

Although we may redefine the ‘structure’ of the common we must not redefine its length. Once a named common block has been set up, each reference to it in a sub-program must be of that size. This does not apply to blank common.

We must be careful with common blocks, especially when we note that anything in a named common block may become undefined when we exit or leave a sub-program using END or RETURN, unless we use the SAVE declaration.

Essentially, this would only apply to the case where we return to a routine which does not have the named common block in it; consider the following:–



Program ANALYS has ‘calls’ to the subroutines PASS1 and COMPUT, and PASS1 has a ‘call’ to subroutine PASS2. This is summarised diagrammatically below:–

The common block B is safe. It is in the main program, so we never return to a routine which does not contain it. But, although subroutines PASS1 and PASS2 can use the common block A, when eventually PASS1 completes its action and returns control to the main routine, i.e. to program ANALYS, the entire common block A becomes undefined, so that if the program should call PASS1 again, those values would be unavailable.

The SAVE statement is a declarative statement which may be used to retain the values in the common block;

```
SUBROUTINE PASS1
COMMON /A/B(100),C(25),L,J,K
SAVE /A/
```

would have the effect of retaining the values of any variables in common block A when we return to the main program. If we used PASS1 again, the values would be available.

In fact, SAVE is more general still, we may use it to retain the values of arrays and variables when a return takes place; otherwise the values would become undefined.

```
DIMENSION X(100)
SAVE X,A,B
```

would retain the values of the array X and the two simple variables.

It is sensible to put variables that are related into the same common block, and to choose a meaningful name.

COMMON is also often used with another important declaration — the DATA statement. The DATA statement is used to provide initial values for variables and arrays; e.g.

```
DATA COEFF1 /10.382/
```

This value of COEFF1 is set up at compile time. It is even more flexible, since we can initialise whole arrays in a very simple way;

```
DIMENSION X(100)
DATA X/100*0.0/
```

This is a form of the implied DO loop. DATA allows us to initialise arrays or simple variables. Variables in DATA statements are only initialised once — either when the program is compiled, or when the program is loaded, depending on the system that you work on. Thus, the DATA statement is not executable, unlike the assignment statement. You may initialise character strings thus

```
CHARACTER STR1*6, STR2*3
DATA STR1/'ABCDEF'/
DATA STR2/'ABC'/
```

Now, you might think that it would be very useful to combine DATA and COMMON statements together, so that you could do something like:

```
COMMON /COEFF1/COEFF1
DATA COEFF1/10.382/
```

but this can only be done in one special program segment — the BLOCK DATA sub-program, and *only* for named COMMON:

```
BLOCK DATA SETUP
COMMON /CAT/X,Y,Z
COMMON /DOG/L
DATA X,Y/1.0,2.0/
DATA Z,L/201.23,3/
END
```

Note that we can initialise any variables that occur in a *named* COMMON statement, but only in this special sub-program which contains no RETURN or STOP statement. This routine is never executed. It is never CALLed or referenced directly in the rest of the program. It is used only at compile time to initialise, and it is not used at run time at all. You may have several BLOCK DATA sub-programs, each with a different name. If you have only one, it need not have a name at all.

COMMON and SAVE are declarative statements, and therefore join the other declaratives at the 'beginning' of the sub-program. DATA belongs to a post-declaration, pre-execution limbo, and may be sandwiched between the two groups. However, it does not matter where the DATA statement comes (provided it is not among the declarative specifications). The DATA causes initialisation at the compile stage, as described above, and its position within the execution statements is irrelevant.

Summary

• Common blocks allow data to be shared between sub-programs.

• Character data *must* be in its own common block.

• The COMMON declaration merely sets aside a section of memory with an identifier (the COMMON block name). You may access this section of memory however you wish in other sub-programs, through the COMMON block name.

• On return to a sub-program, the contents of a common block may become undefined. Blank common never becomes undefined within a program. The SAVE declaration will safeguard named COMMON.

• Data may be initialised in DATA statements; these are not executable statements, but are set up at ‘load’ time.

• You may not use the DATA statement to set up the contents of a COMMON block, except in the BLOCK DATA sub-program.

Problems

1. Write a program that uses a DATA statement to initialise an integer variable. This integer variable must be in a common block. The program should contain a main program, block data sub-program, and two subroutines. The main program should print out the value of the integer variable initialised in the DATA statement. Each of the two subroutines should also print out the value of this same variable. Each subroutine just prints out the value of the integer variable. The main program should

- print out the value of the variable
- call subroutine 1
- print out the value of the variable again.
- call subroutine 2
- print out the value of the variable again

What do you notice about the values printed out? Alter the variable to be of type character, and run the program again. You will need to alter the DATA statement also. What do you notice now?

2. Some of the previous programs, subroutines and functions could benefit from using the DATA statement to initialise some information. For example, the coefficients of the Bessel function in the problems in Chapter 12 could usefully be initialised in this way (either as simple variables, or as vectors in an implied DO). Do this.

21

Optimisation

We may define our basic attitude to optimization in two rules:

Rule 1: Don’t do it.
Rule 2: Don’t do it yet.

M. A. Jackson, ‘Principles of Program Design’

Aims

The aims of this chapter are:–

- to introduce some reasons for **NOT** optimising a program
- if the reasons are not sufficient then to show some ways in which you *can* optimise a program

Introduction

Optimisation is rarely done for the right reasons. As you are now aware, writing programs is not easy, and re-writing a program in such a way as to make it ‘faster’, but less easy to comprehend, is dangerous. This is because finding bugs in the program will now become much harder, and any time gained by a shortening of the computer’s time will be more than offset by extra effort on your part.

The first thing to consider is whether the solution to the problem is the most appropriate. Sometimes this is not that obvious. Consider the case of a physicist and mechanical engineer approaching the problem of designing a combustion engine independently of one another. There will be knowledge at the disposal of the physicist that is not at the disposal of the engineer and vice versa. Each will solve the problem in a different way and neither is ‘correct’; both are appropriate depending on the exact requirements of the problem. So there may be a more appropriate way of solving the problem. The thing to consider is whether the problem can be solved in another way. Human beings are very reluctant to throw away something that they have done and start again. You must force yourself to do this if circumstances demand it.

Assuming that you do not have to redesign the whole solution, the next thing to do is find out where in your program most time is spent. There should be profiling tools available to give you some idea here. Lacking adequate tools one should make an educated guess. Note that, this guess may be totally erroneous, and, if you are not sure, then obtain some help.

It may turn out at this stage that the program spends most of its time in the Fortran run time systems or in operating systems routines. If this is the case, you have two options; firstly to abandon the project, or secondly consider how you can improve on the Fortran run time library, or operating system. This is not an easy task. You may well have to learn a considerable amount about an area which is completely unrelated to your problem.

Let us assume now that we have identified the problem as being in your program. What do we do next?

Some of the ‘improvements’ relate to the structure of the computer, but a few are sufficiently general that they should be borne in mind when programming:

- eliminate redundant instructions and expressions within statements, and within sequences of statements
- evaluate loop invariant expressions outside the loop
- where a subscripted variable is used several times, equate it to a temporary variable

Different operations take different times on a computer. This table (from Heath and Meek, 1979) gives some rough notions of the relative speeds of operations:

Operation	Relative speed	Example
integer assignment	1	I=10
integer addition/subtraction	1.5	I+J
real assignment	2	A=10.0
real add/subtract	3	A+B
real multiply	5	A*B
integer to real	6	A=I
integer multiply	8	I*J
division	9	I/J A/B
exponentiate to integer	35	I**J
exponentiate to real	115	A**B
		I**B

This implies that if you are interested in making your program efficient, you should, for example replace the following operations by these others.

Original		Replacement
2.0*X		X+X
X/10.0		X*.1
X**2.0		X**2
	or	X*X

Similarly, reducing the numbers of operations by introducing temporary variables will also have an effect:

```
Z=X*Y+W/(X*Y)**2.5
```

could be replaced by

```
XY=X*Y
Z=XY+W/XY**2.5
```

An even more striking example, which demonstrates how a polynomial evaluation can be done without any exponentiation, is the following:

```
        Y=A(0)*X**5+A(1)*X**4+A(2)*X**3+A(3)*X**2+A(4)*X+A(5)
```

can be replaced by:

```
        Y=((((A(0)*X+A(1))*X+A(2))*X+A(3))*X+A(4))*X+A(5)
```

At a cruder level, simple recourse to algebra can simplify expressions:

```
        A=B*(E+F)-C*(E+F)+D*(E+F)
```

reduces to

```
        A=(B-C+D)*(E+F)
```

when the common factor is removed.

Remember that you can sometimes make one DO loop do lots of things, and thus:

```
        DO 1 I=1,100
            A(I)=B(I)*C(I)+4.
1       CONTINUE
        DO 2 J=1,100
            D(J)=E(J)+5.
2       CONTINUE
```

is equivalent to, but not as fast as

```
        DO 1 I=1,100
            A(I)=B(I)*C(I)+4.
            D(I)=E(I)+5.
1       CONTINUE
```

If you are using multi-dimensioned arrays, write the DO loop referencing so that the ‘first’ index varies first:

```
        DIMENSION X(25,5,200,100)
        DO 1 I=1,100
            DO 2 J=1,200
                DO 3 K=1,5
                    DO 4 L=1,25
                        X(L,K,J,I)=0.0
4                   CONTINUE
3               CONTINUE
2           CONTINUE
1       CONTINUE
```

This relates to the way in which the array is stored in the computer’s memory. You might also note that ‘short’ DO loops are sometimes not very efficient, and that it would be better to rewrite

```
      DO 1 J=1,N
         DO 2 I=1,3
            X(I,J)=Y(I,J)+Z(J,I)
2        CONTINUE
1     CONTINUE
```

as

```
      DO 1 J=1,N
         X(1,J)=Y(1,J)+Z(J,1)
         X(2,J)=Y(2,J)+Z(J,2)
         X(3,J)=Y(3,J)+Z(J,3)
1     CONTINUE
```

even though this would take longer to write out.

Sometimes it is possible to restructure the problem slightly. In the following example, the IF test is carried out every time round the loop;

```
      DO 1 I=1,100
         IF(FLAG)THEN
            A(I)=B(I)-2.*C(I)
         ELSE
            A(I)=B(I)+3.*D(I)
            B(I)=X*D(I)
         ENDIF
1     CONTINUE
```

we could replace this by

```
      IF(FLAG)THEN
         DO 1 I=1,100
             A(I)=B(I)-2.*C(I)
1        CONTINUE
      ELSE
         DO 2 I=1,100
             A(I)=B(I)+3.*D(I)
             B(I)=X*D(I)
2        CONTINUE
      ENDIF
```

In the latter case, the test is done only once.

The next example shows how the Fortran might be coded, straight from a set of equations:

```
        DO 1 K=1,M
            B(K)=0.0
            DO 2 J=0,N-1
                A(K)=A(K)+X(J)*COS(J*K*DELTAY)
                B(K)=B(K)+X(J)*SIN(J*K*DELTAY)
2           CONTINUE
            A(K)=A(K)*2./(N-1)
            B(K)=B(K)*2./(N-1)
1       CONTINUE
```

With a little thought, this could have been rewritten as:

```
        N1=N-1
        C1=2./N1
        DO 1 K=1,M
            AK=0.0
            BK=0.0
            DO 2 J=0,N1
                YJ=J*K*DELTAY
                XJ=X(J)
                AK=AK+XJ*COS(YJ)
                BK=BK+XJ*SIN(YJ)
2           CONTINUE
            A(K)=C1*AK
            B(K)=C1*BK
1       CONTINUE
```

These are relatively minor improvements which can be made, without the more major invocation of either recurrence relationships or an FFT (Fast Fourier Transform). In the first example there are 6*M*N+4*M multiplications and/or divisions, while in the second there are 4*M*N+2*M+1.

Summary

It may be that one problem solution is computationally more efficient than another, but human efficiency is also important, and it is almost always better to be slow and ‘correct’ than efficient and wrong. The computer is supposed to be working for you, not you for the computer.

There are some simple rules which can be adopted which do not destroy the comprehensibility of the program steps, and will reap some benefit, especially if they are incorporated at the earliest stages of program development.

Problems

1. Write a program to sort 5000 numbers. The numbers should be stored in an array. One way of obtaining the 5000 numbers is through a random number generator. Most random number generators are machine specific, but you should have access to one through Fortran. Likely names are RANF, RAND,

RND and so on. Other machine dependent functions will return central processor time used — this would provide an objective figure for any optimisation you achieve.

Now apply the guidelines for optimisation given in this chapter. What difference do they make?

Now get hold of a book on sorting and searching and use one of the recommended algorithms e.g. quick-sort. What difference does this make? How much time have you spent so far?

Now use one of the standard subroutine libraries available on your system. What improvement have you got now?

Now use the SORT package available on your machine. What time did this package take?

Was it worth it?

2. Generating prime numbers is a favourite task for many mathematicians. The Collected ACM Algorithms contain several examples of programs which will calculate the first k prime numbers (e.g. Algorithms 35, 310 and 311). If you have access to these algorithms, compare them and read the accompanying remarks. Essentially it appears that the running time to compute the first k primes is of the order k**n, where n may be as small as 1.35. If you do not have access to the Collected ACM Algorithms, consult Knuth, *Fundamental Algorithms,* on the same subject.

Now answer these questions; why would you wish to create a table of the first k primes more than once; would it be easier (and more ‘efficient’) to store the table on file than to recalculate it?

22

Problem solving revisited

As it was, their judgement was based more on wishful thinking than on sound calculation of probabilities;

Thucydides, 'The Peloponnesian War'

Aims

The aims of this chapter are to draw together some of the ideas that have been presented regarding problem solving. As with many situations where new concepts are involved some experience at a concrete level is required before the ideas really make sense.

Introduction

It should be obvious by now that the intellectual skills involved in programming are not to be underestimated. Part of the reason lies in the nature of the tasks that we are asking the computer to perform. These are typically many orders of magnitude more complex than the human mind can perform unaided. Part of the reason lies with the statement of the problem solution in a programming language.

There are two things to consider here. Firstly we need help at the design stage to generate a possible solution, i.e. we need to adopt proven methods of working which will make the design stage easier. Some of the methods that we can adopt have been refined over many years and go back to the Greeks, e.g. Euclid. Others will be more recent and are developments based on experience over the last 30 years of programming.

Secondly we need help at the coding stage to try and make our programs more easily understandable. This is especially important as the programs that we write get more and more complex as the problems that we undertake become more ambitious.

These two parts are interconnected, and should not be regarded in isolation. We need to have an idea about both if we are to become proficient programmers. Let us now consider each in turn.

Algorithms

The name algorithm is derived from al-Khowarazmi, who was an Arab mathematician who wrote a treatise on algebra around 830 AD. There are many definitions of algorithm to be found in computing books, but the one given at the start of this book is sufficient for our purposes. i.e. a sequence of operations that will solve part or all of a problem. The next thing to consider is the way in which we can break problems down into sequences of operations. A more formal discussion on algorithms can be found in Korfhage, *Logic and Algorithms.*

Abstraction

The most powerful technique that we have at our disposal is that of abstraction. This means that we can hide the complexity of what we are doing by using a term or phrase like ‘invert a matrix’ and concentrate on the result rather than how the action will be performed. This means that we are able to postpone the fine detail of each step of the solution and concentrate on one aspect of a problem at a time. The importance of abstraction will become obvious when we consider the next sections. You should already be familiar with the idea of abstraction from your discipline when you can use the results of someone’s work without having to actually understand all aspects of it. Further reading on the subject of abstraction can be found in Dahl, Dijkstra and Hoare, *Structured*

Programming; in Brinch-Hansen, *Operating System Principles;* and in Wulf's contribution in *Current Trends in Programming Methodology.*

Structured programming

The main concerns of structured programming are firstly with the ways in which we can reduce the complexity of a problem and achieve a solution, and secondly with the correctness of the solution. As can be appreciated from the last section abstraction has an important contribution to make here. Let us consider the first aim which is the reduction of the complexity of the problem.

This can be achieved by breaking the problem down into parts. Most programmers overestimate their ability to cope with complexity. They often write large monolithic programs with little apparent structure. This means that is difficult to predict the action of the program or parts of the program. Consider the following, which is based on actual experience of the authors whilst working in an advisory capacity.

> We are trying to understand a part of a program, let us say section A. In the middle of section A there is a jump to another part of the program, section B. So to understand how section A will work we need to examine section B. There is a jump in section B to another part of the program, section C. Now to predict how section A works we need to find out how both B and C work.

It should not be imagined that the above is at all unusual. A program may grow in complexity over several months and may be required to perform many tasks not in the original definition and design. Thus when we start programming we must develop habits that will allow us to retain mastery in situations like this. Let us now consider what we can do concretely to achieve this mastery.

Firstly we use our powers of abstraction to hide the complexity of what we are trying to do. We do this by designing and specifying actions in general terms, and concentrate on the results of the actions rather than the how of the actions. We gradually refine each of these actions until we are talking in terms of actual 'code'. We structure our solution into small pieces so that we can say with certainty that this section of code will do exactly what we want and no more. We will consider this in more detail in a later section paying particular attention to the impact this has on structures in programs.

Let us now consider how we can insure the correctness of our program. We can achieve this by making our programs understandable. We should aim to produce programs that bridge the gap between the subject area specification, and the solution in a programming language. This is not easy, as a programming language may have no adequate base constructs in many cases. Thus we must create our own. Consider the problem of a payroll program where the rate of pay is different on Saturday and Sunday. We have to represent the concept of days of the week in terms of the types of variables at our disposal. In Fortran we may choose integers, in the range 1 to 7 say, or we could choose characters, and actually use the strings ‘Monday’ etc. to represent the days. Neither of the above approaches will stop us in the first case assigning a value of 8 to the integer variable, and an inappropriate string to the character variable. Note in the latter case that we may misspell the string and have ‘Mnoday’. The severity of this problem will vary with the programming language that you use. Fortran has only a few base types, and it is unreasonable, for example, to expect to write large database applications in the language. Pascal, on the other hand, will allow you to define your own types of variables, and thus is suited for many applications that Fortran is not. Thus you must be careful of the kind of problem that you try to solve with a particular programming language. It may be quicker to learn another programming language, rather than force a solution in a language that was not designed to cope with that particular kind of problem. Remember that we are trying to achieve as close a correspondence as possible between the problem and its solution in a programming language. If we strive for this then we will make fewer errors.

Let us now consider what we can do in practical terms to achieve the above. Firstly we can adopt a small set of program control constructs. i.e. we work with a small set of sequencing mechanisms. These forms are:–

- sequences of operations, or concatenation
- alternatives between courses of actions
- loops, or repetition of statements
- sequential flow

As a Fortran program is sequential, this is satisfied fairly easily.

Alternatives

This has generated considerable heat in the computing world. The divide is between the people who restrict themselves to proven forms, and the school who want no restrictions on the way they work. There is considerable argument in the ‘real’ world about the use of crash helmets when riding a motor-bike. We leave you to draw your own conclusions. Let us now consider some of these forms:–

- The IF THEN ENDIF construct. This enables to choose a course of action if necessary. Note that we continue the sequential flow after the execution of the if block.
- The IF THEN ELSE construct. This allows us to choose between two courses of action before continuing with the normal sequential flow.

- The IF THEN ELSEIF construct. This allows us to choose between many possible courses of action. This is sometimes given the name a *case* construct — as we are choosing between several cases.

These are sufficient to handle most problems.

Loops

Repetition can be handled by three basic forms. These are:–

- The WHILE construct. This does not exist as a base create in Fortran. Therefore we create it from more primitive forms. These are the IF and GOTO.
- The REPEAT UNTIL construct. Again this does not exist in Fortran as a base form. We must construct it also from the IF and GOTO.
- The simple loop controlled by a counter or index. This is programmed in Fortran with a DO loop.

The above are representative of current thinking in programming. It is possible that others may be developed in the future, but for the present it is recommended that you restrict yourself to these. If there are revolutionary constructs and ideas waiting to be discovered in the future you can be sure that you will hear of them eventually.

Structure in data

So far we have only considered structure in the program. A program manipulates data, and there is generally structure to the data. To quote Wirth

Algorithms + data structures = programs

Thus one should look to see what structure exists in the data. You are already familiar with the array as a data structure. You have also seen that tables of data exists in many problems. There are only a few data structures in Fortran, but they are sufficient for a large number of applications. It may be necessary in many problems to reduce the real data structure to one that can be represented in the language we are using. The array is one of the most fundamental data structures. It is generally possible to transform a data structure into something that can be manipulated by an array, e.g. vectors, lists, stacks etc. The bibliography contains references to several books which emphasise this point.

Top Down and Bottom Up

We are now in a position to consider these two approaches. It should be apparent that top down design is going to be of much more use than bottom up.

However there will be instances where you have no idea where to start. Then you consider what you can do, and work backwards to a possible solution. This technique will be familiar to some readers as it is similar to a technique sometimes used in mathematics, i.e. we work backwards from what we want to where we are. However the analogy should not be stretched too far.

Step wise refinement

This term should now mean something. We use this technique to work gradually towards a solution from the problem definition. It is closely linked with the idea of top-down design.

Modular programming

We achieve modularity in programming in Fortran using functions and subroutines. These enable us to construct sets of actions and we can put these actions together to solve our problem. What has been missing so far in this chapter is the way in which we allow the communication to take place between these modules. This brings us to the concept of localised action. We are interested in ensuring that when we use a function or subroutine it will do what we want and no more. Care must be taken therefore when designing functions and subroutines so that, whilst they are sufficiently general that we can use them in several ways, we do not want to make them so complex to use that they may have unsuspected side effects. Functions are invoked by name and communication is generally restricted to take place through its arguments. Subroutines pose a few problems. Communication here can take place through arguments, and often also through common blocks. To keep control of the complexity it is therefore recommended that the number of channels of communication is kept to a minimum, to enable us to understand fully what is happening.

Concluding remarks

There are a few drawbacks in the above approaches. The main one is that the program rarely contains any information about the decisions that took place at the design stage, i.e. there is no information on the abstraction process, or what went on whilst testing and debugging the program.

This can be a tremendous problem when you come to modify the program. It is therefore recommended that you get into the habit of inserting comments into the program about the design process. This includes putting in comments about errors in your thinking where relevant. You should also put in comments about alternatives that you rejected. This may enable you to develop a better approach the next time you tackle a problem. It is the experience of the authors that explicitly writing down your thinking, or even articulating it to another person, can be used to great benefit in exposing the flaws in your logic. This is especially true when you start programming. Thus working in small groups at the start can have a profound impact on the time spent programming.

Lastly, a plea which may go unheeded initially. Many people see programming as an extension of their own personality. Since the program is their own creation, any criticism is seen as a personal criticism. Try to rise above this. If your ego can be separated from your program, you will not only find it easier to seek and find advice, but you will also avoid ulcers, and keep friends. Your programs might improve too. A criticism of your program is not a criticism of you.

23

Operating systems

‘Mow your lawn, lady’

James Blish, ‘Cities in Flight’

Aims

The aims of this chapter are:–

- to give a brief historical review of the development of operating systems
- to note the impact of present day operating systems on the process of program development
- to glance at other developments in computing which are likely to have an impact in the future on this process

The operating system

Most computer systems provide an operating system. These will vary considerably from small systems available on personal computers (e.g. MS/DOS, PC/DOS) through to very complex systems available on mainframes (e.g. IBM's VM system). The importance of the operating system should not be underestimated. They greatly influence a user's view of a computer system. The purpose of this chapter is to provide some background information on the changes that have taken place over the past 40 years with operating systems. The relevance of this is that operating systems will change during the time that you spend computing, and thus is it is important to have some perspective on these developments. Operating systems can help or hinder the task at hand, and it is important to get the system to help you, and maximise your productivity. After all the computer is a tool for your use.

The 1940s

In the early days of computing there were no operating systems. The user had complete access to the whole of the machine. This meant that programmers were involved in areas which were nothing to do with the problem that they were interested in.

The 1950s

As you may imagine this was not very satisfactory. The next development that took place was the introduction of batch operating systems. The main impact was that the user was distanced from the machine. Greater use of an expensive resource was also made.

The 1960s

The next development was multiprogramming. This enabled the computer to have several jobs under execution and to switch between them as resources were available. Thus timesharing soon became possible — users could now communicate through a keyboard. This had a considerable impact on the program development process.

The 1960s to 1970s

This era saw the development of large multi-purpose operating systems. They had to be able to cope with a wide variety of demands. The most famous was the IBM System 360. Their development represented some of the most ambitious programming projects ever undertaken. An amusing and instructive discussion of the problems encountered developing the IBM system is given in Brooks, *The Mythical Man Month*. A lot was learned from the failures of the development of these operating systems.

One notable success from this era was the UNIX system. The development of the UNIX system is a very interesting way of approaching the design of an operating system. Further information on the UNIX system can be found in the references in the bibliography.

The 1970s to today

The operating systems of this era are primarily refinements and extensions to the ones of the previous era. See Deitel, *Operating Systems*, for more information on this subject.

Other developments

There have been developments in other areas of computing which have had greater user impact in this era. The four areas of most interest are:–

- the availability of cheap and powerful microprocessors
- the increasing use of computer networks
- the development of parallel processing hardware
- the introduction of alternate mechanisms for interfacing to the computer

The development of cheap and powerful micro-processors has meant that many tasks can now be done locally, using a micro-processor, rather than using a central system. Combined with the use of networks this often means that the end user is unaware of exactly where the 'computing' is taking place. In a network you will often have a choice of several machines, from your own terminal, and each machine may support only a small part of the whole service available.

Parallel processors help to remove the strict sequential nature inherent in many programming languages. A simple example is adding two vectors together. In a conventional processor, each element would be taken one at a time. On a parallel processor it could be possible to do the whole operation at once. The widening availability of parallel processing hardware will have an impact in two areas. The first concerns the narrow idea of similar operations on data, e.g. multiplication of arrays, and the second concerns the development of suitable algorithms for these machines. Of these the second seems the most challenging area.

The major device for communicating with a computer is a typewriter keyboard. The cost of alternatives (e.g. graphics tablets, touch sensitive screens, mice) has been prohibitive and has been restricted to a small percentage of the user community. The development of cheap and powerful micro-processors has had a significant impact in this area. Research and development work in this area has been going on for some time and it is only recently that this development work is becoming available to a wider population (cf. Goldberg and Robson, 1984). Communication is now possible through ‘touch sensitive’ *vdu* screens, joystick, trackball or mouse controls, voice activation, as well as more mundane means.

24

Tools in programming

Man is a tool-making animal.

Benjamin Franklin.

Man is a tool-using animal...
Without tools he is nothing
With tools he is all.

Thomas Carlyle.

Aims

The aims of this chapter are:–

- to introduce the idea that there are other programs that can help in the program development process
- to give some examples of the sorts of programs that are available

Introduction

When you become involved in programming on a regular basis it is worthwhile considering what programs are available on the computer system. You have already used an editor and a compiler, but these represent a small subset of what is likely to be available. The programs that are discussed in this chapter are given the name 'tools'. These are programs that can be used to save time and effort when involved in developing your own programs. They are given the name tools because they serve a similar purpose to 'tools' that you use in carpentry, brick laying, engineering etc. You may have encountered the phrase 'software engineering' and can thus see the origin of the word 'tool' in this sense.

The following tools were available on a number of systems the authors have used. The aim here is to show you some of the system for examples of the tools they provide.

Update

This is a very sophisticated tool that is used in the development and maintenance of source programs. At the simplest level it allows you to keep track of program changes. As with most good tools it can be used at a variety of levels, and a complete understanding of a tool like this may well take some years.

Compare

This tool shows the differences between two text files, on a line by line basis. It can be used to compare any text files on the computer system. It can be used in conjunction with Update to compare two versions of a program, and give a list of the changes necessary to create one from the other.

Indent/Pretty Print

A tool for indenting Fortran programs. This is a very useful tool for 'tidying up' and making the structure of Fortran programs clearer. The program will indent loops etc., making the program much easier to read and understand.

Sort/Merge

A tool for sorting and merging files. It is a very common requirement that data will need to be sorted in some way before it can be used by another program.

The above tools exist on the system that the authors work on. Similar tools should exist in some form or another on your system.

The load process and the loader

When you compile a program, a file is generated. you work on, e.g. relocatable, or object file, or binary file. The key thing is that the file does not contain sufficient information for the hardware to execute it in its present form. Typically there is some linking required to a library of routines supplied by the manufacturer, e.g. if you use the square root function there will be linking at load time to the routine that evaluates square roots. This routine has been written by the manufacturer, and will have already been compiled. It will exist in a library of precompiled routines often called the Fortran run time library. On some machines you will have been made aware of this, but on most timesharing systems this process will have been hidden by the provision of a sophisticated system program called a loader, or link loader. There will be manuals describing how to use and control this program on your system, and a chat with some of the support staff of your installation is generally the best place to start.

Compiled routines and library maintenance

There will also be tools for the maintenance of compiled routines. These have been given the name *libraries* in this book. There will be tools to create, modify and update libraries. On many systems these tools are very sophisticated and there will be a large investment of your time required before you master their full capability. There will be ways of controlling the load process so that other system libraries and user libraries can be ‘searched’ or included in the load process. These libraries may contain routines to calculate standard statistical functions, plot graphs etc.

Job control languages

You get the system to do what you want by typing in certain commands — e.g. ED may invoke the editor. There are many commands available on a computer system. They are often called operating systems commands. It is worthwhile finding out some of the commands provided on your system. There will be both manufacturer and installation supplied commands.

On many systems it is possible to group these commands together so that it will be possible to invoke a sequence of operating systems commands by typing in one name. This ‘command’ file is a ‘text’ file and may be created using an editor.

It is also possible to vary the action of these command files by the provision of elementary programming constructs within the command file.

- Set and test system and job-control variables. This will enable us to test for the existence of files for example, and take appropriate action.

- Loops can be set up so that groups of commands can be repeated, e.g. several sets of data, possibly from a magnetic tape, could be analysed in a similar fashion.
- Arguments can be passed from one command file to another, e.g. number of data sets, file names or magnetic tape identifiers.
- It is possible for one process to create another. Thus you may set one command file executing, and depending on the progress of this activity another process may be generated, e.g. a job may run which creates a data file, and you can set up another job to archive this data file onto magnetic tape.

These are only some of the things that can be done with a job control language. A good job control language can be regarded as a primitive programming language, and the investment in learning about the capabilities of your job control language will generally repay itself fairly quickly.

Program development systems

Operating systems are developed for a variety of reasons. One of these may be the provision of a system with facilities that aid in the program development process. One of the operating systems that has become widely accepted for this purpose is UNIX. If you have access to a UNIX system locally then it will be worthwhile actually going and talking to some of the users of this system. The book *The UNIX System* by Bourne gives some idea of the capabilities of the system.

TOOLPACK represents a collaboration between America and Britain in the provision of a suite of portable tools and it is hoped will become widely available in both countries. One of the stated purposes of TOOLPACK is the provision of a strong, comprehensive tool system for programmers who are producing, testing, transporting, or analysing moderate size mathematical software written in Fortran.

Summary

It is not the intention to offer a definitive statement on tools here, rather to present some of the ways in which you can make effective use of a computer system. There is a considerable amount of software already written, tested, and documented for most computer systems. It is well worth the effort finding out about this software from local users. It may be that you will never need to write programs yourself, but always find something that can be used to satisfy your own particular requirements. However there is considerable satisfaction to be gained from the production of an easy to use, well tested and documented piece of software.

Notes

Chapters 2 and 3 have their own self contained bibliographies, and should be consulted for references on problem solving and programming languages.

M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions,* Dover, 1970

> This book contains a fairly comprehensive collection of numerical approximations for many mathematical functions, of varying degrees of obscurity. It is a widely used source.

ANSI X3J3, *Programming Language FORTRAN,* American National Standards Institute, 1978

> This is the book that defines the standard for Fortran. It is interesting to read parts of the book and see how difficult it is to make English totally unambiguous.

Association for Computing Machinery, *Collected Algorithms*, 1960–1974, and *Transactions on Mathematical Software,* 1975–,

> A good source of some rather specialised algorithms. Early algorithms tend to be in Algol, but Fortran predominates now.

The Bell System Technical Journal, *Unix Time–Sharing System,* July August 1978, Vol.. 57, No. 6, Part 2.

> A collection of papers from the original team that designed, implemented and used UNIX. Extremely interesting history and insight into the UNIX system.

S. R. Bourne, The UNIX System, Addison-Wesley, 1982

> A comprehensive coverage of the facilities provided by the UNIX system.

Per Brinch–Hansen, *Operating System Principles*, Prentice-Hall, 1973

> An ‘old’ but interesting book on operating systems. Also contains some brief comments on problem solving.

F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1974

The book is a collection of essays by one of the people responsible for the development of the IBM/360 operating system. The book is very readable, and amusing in parts. It is recommended reading for anyone involved in programming on a regular basis.

O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972

This is the seminal book on structured programming.

H. Deitel, *Operating Systems*, Addison-Wesley, 1984

The book gives a comprehensive coverage of most aspects of operating systems. The book also contains several case studies of current operating systems.

M. Elsen, *Concepts of Programming Languages,* Science Research Associates, 1973

Reasonable introduction to various apsects of programming languages.

A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, 1984

The book presents some of the ideas and concepts involved in communicating with a machine using a sophisticated graphical, interactive, programming environment.

P. Heath, and B. Meek, *Guide to Good Programming Practice*, Ellis Horwood, 1979

Contains much practical advice for the beginner (and the not so beginner).

R. Hunt and J. Shelley, *Computers and Common Sense*, Prentice Hall, 1983

Provides a good introduction to many aspects of computing.

M. A. Jackson, *Principles of Program Design*, Academic Press, 1975

In this book, Jackson describes a very structured and professional approach to large scale computer software projects. Much of what he says is also applicable at a smaller scale.

H. Katzan, *Fortran 77*, Van Nostrand Reinhold, 1978

A reasonably readable (and far more compact) description of the Fortran 77 language.

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976

Interesting ‘essays’ on the program development process.

Donald E. Knuth, *The Art of Computer Programming*, Addison-Wesley,

Vol.1 *Fundamental Algorithms*, 1974
Vol.2 *Semi-numerical algorithms*, 1978
Vol.3 *Sorting and searching*, 1972

Contains interesting insights into many apsects of algorithm design. Good source of specialist algorithms. Knuth writes with obvious and infectious enthusiasm (and erudition). He may yet write the definitive computer novel.

R. Korfhage, *Logic and Algorithms*, Wiley, 1966

A more formal and rigorous introduction to algorithms.

M. Metcalf, *Fortran Optimization*, Academic Press, 1982

A very useful book for anyone wishing to optimize Fortran, especially on large machines. Contains much which is relevant generally. An added bonus is the program INDENT, included in the book, which will take a Fortran 77 program and indent DO loops and Block If statements.

Numerical Algorithms Group, *FORTRAN Library Manual*, Mark 13 (several volumes), NAG, 1989

Description and definition of the Fortran interface for the NAG library. Contains descriptions of the techniques used and example programs.

Adrian Oldknow and Derek Smith, *Learning Mathematics with Micros*, Ellis Horwood, 1983

> Although directed towards the programming language BASIC, this book contains many useful little algorithms, and is very concerned with discussing the reasons behind the programming.

Leon J. Osterweil, Toolpack – *An Experimental Software Development Environment Research Project*, IEEE Transactions on Software Engineering, Vol. SE–9, No. 6, pp.673–685, November 1983

> Discusses the goals and methods of the Toolpack project, providing some notion of its scope. This software is initially ‘public domain’, and is likely to be widely available in the academic and research communities.

Ralston A. and Rabinowitz P., *A First Course in Numerical Analysis,* 2nd Edition, McGraw Hill, 1978.

> One of the classic numerical analysis textbooks.

W. A. Watson, T. Philipson and P. J. Oates, *Numerical Analysis,* Arnold, 1981

> Subtitled *The Mathematics of Computing* this book provides a good account of the problems which arise with limited machine precision, and some of the solutions which are possible.

G. H. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971

> Has interesting comments to make about the psychology of the programmer. The originator of the term ego-less programming.

N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976

> Good presentation of the ideas involved in the discipline of computer programming.

Raymond Yeh (Editor), *Current Trends in Programming Methodology, Software Specification and Design*, Prentice Hall, 1977

> Contains several stimulating papers on the design process.

S. J. Young, *Real Time languages, design and development*, Ellis Horwood, 1982

> The first part of the book contains a reasonable coverage of some of the ideas involved in the design of a programming language.

0	nul	32		64	@	96	‘
1	soh	33	!	65	A	97	a
2	stx	34	"	66	B	98	b
3	etx	35	#	67	C	99	c
4	eot	36	$	68	D	100	d
5	enq	37	%	69	E	101	e
6	ack	38	&	70	F	102	f
7	bel	39	’	71	G	103	g
8	bs	40	(	72	H	104	h
9	ht	41	)	73	I	105	i
10	lf	42	*	74	J	106	j
11	vt	43	+	75	K	107	k
12	ff	44	,	76	L	108	l
13	cr	45	-	77	M	109	m
14	so	46	.	78	N	110	n
15	si	47	/	79	O	111	o
16	dle	48	0	80	P	112	p
17	dc1	49	1	81	Q	113	q
18	dc2	50	2	82	R	114	r
19	dc3	51	3	83	S	115	s
20	dc4	52	4	84	T	116	t
21	nak	53	5	85	U	117	u
22	syn	54	6	86	V	118	v
23	etb	55	7	87	W	119	w
24	can	56	8	88	X	120	x
25	em	57	9	89	Y	121	y
26	sub	58	:	90	Z	122	z
27	esc	59	;	91	[	123	{
28	fs	60	<	92	\	124	|
29	gs	61	=	93	]	125	}
30	rs	62	>	94	^	126	~
31	us	63	? l	95	_	127	del

YET IF HE SHOULD GIVE UP WHAT HE HAS BEGUN, AND AGREE TO MAKE US OR OUR KINGDOM SUBJECT TO THE KING OF ENGLAND OR THE ENGLISH, WE SHOULD EXERT OURSELVES AT ONCE TO DRIVE HIM OUT AS OUR ENEMY AND A SUBVERTER OF HIS OWN RIGHTS AND OURS, AND MAKE SOME OTHER MAN WHO WAS ABLE TO DEFEND US OUR KING; FOR, AS LONG AS BUT A HUNDRED OF US REMAIN ALIVE, NEVER WILL WE ON ANY CONDITIONS BE BROUGHT UNDER ENGLISH RULE. IT IS IN TRUTH NOT FOR GLORY, NOR RICHES, NOR HONOURS THAT WE ARE FIGHTING, BUT FOR FREEDOM - FOR THAT ALONE, WHICH NO HONEST MAN GIVES UP BUT WITH LIFE ITSELF.

QUEM SI AB INCEPTIS DIESISTERET, REGI ANGLORUM AUT ANGLICIS NOS AUT REGNUM NOSTRUM VOLENS SUBICERE, TANQUAM INIMICUM NOSTRUM ET SUI NOSTRIQUE JURIS SUBUERSOREM STATIM EXPELLERE NITEREMUR ET ALIUM REGEM NOSTRUM QUI AD DEFENSIONEM NOSTRAM SUFFICERET FACEREMUS. QUIA QUANDIU CENTUM EX NOBIS VIUI REMANSERINT, NUCQUAM ANGLORUM DOMINIO ALIQUATENUS VOLUMUS SUBIUGARI. NON ENIM PROPTER GLORIAM, DIUICIAS AUT HONORES PUGNAMUS SET PROPTER LIBERATEM SOLUMMODO QUAM NEMO BONUS NISI SIMUL CUM VITA AMITTIT.

from *‘The Declaration of Arbroath’* c.1320. The English translation is by Sir James Fergusson.

OH YABY NSFOUN, YAN DUBZY LZ DBUYLTUBFAJ BYYBOHNX GPDA FNUZNDYOLH YABY YAN SBF LZ B GOHTMN FULWOHDN DLWNUNX YAN GFBDN LZ BH NHYOUN DOYJ, BHX YAN SBF LZ YAN NSFOUN OYGNMZ BH NHYOUN FULWOHDN. OH YAN DLPUGN LZ YOSN, YANGN NKYNHGOWN SBFG VNUN ZLPHX GLSNALV VBHYOHT, BHX GL YAN DLMMNTN LZ DBUYLTUBFANUG NWLMWNX B SBF LZ YAN NSFOUN YABY VBG YAN GBSN GDBMN BG YAN NSFOUN BHX YABY DLOHDOXNX VOYA OY FLOHY ZLU FLOHY. MNGG BYYNHYOWN YL YAN GYPXJ LZ DBUYLTUBFAJ, GPDDNNXOHT TNHNUBYOLHG DBSN YL RPXTN B SBF LZ GPDA SBTHOYPXN DPSENUGLSN, BHX, HLY VOYALPY OUUNWNUNHDN, YANJ BEBHXLHNX OY YL YAN UOTLPUG LZ GPH BHX UBOH. OH YAN VNGYNUH XNGNUYG, YBYYNUNX ZUBTSNHYG LZ YAN SBF BUN GYOMM YL EN ZLPHX, GANMYNUOHT BH LDDBGOLHBM EN-BGY LU ENTTBU; OH YAN VALMN HBYOLH, HL LYANU UNMOD OG MNZY LZ YAN XOGDOFMOHN LZ TNLTUBFAJ.

A02	Complex arithmetic
C02	Zeros of polynomials
C05	Roots of one or more transcendental equations
C06	Summation of series
D01	Quadrature
D02	Ordinary differential equations
D03	Partial differential equations
D04	Numerical differentiation
D05	Integral equations
E01	Interpolation
E02	Curve and surface fitting
E04	Minimising or maximising a function
F01	Matrix operations including inversion
F02	Eigenvalues and eigenvectors
F03	Determinants
F04	Simultaneous linear equations
F05	Orthogonalisation
F06	Linear Algebra Support Routines
G01	Simple calculations on statistical data
G02	Correlation and regression analysis
G04	Analysis of variance
G05	Random number generators
G07	Univariate Estimation
G08	Nonparametric statistics
G11	Contingency Table Analysis
G13	Time series analysis
H	Operations research
M01	Sorting
P01	Error trapping
S	Approximations of special functions
X01	Mathematical constants
X02	Machine constants
X03	Innerproducts
X04	Input/Output utilities

Name	Description	Arguments No.	Type	Result Type	Example
INT	Converts to integer from integer, real, double precision and complex	1	I, R, DP, C	I	I=INT(R)
REAL	Converts to real from integer, real, double precision and complex	1	I, R, DP, C	R	R=REAL(I)
DBLE	Converts to double precision from integer, real, double precision and complex	1	I, R, DP, C	DP	D=DBLE(R)
CMPLX	Converts to complex from integer, real double precision and complex	2	I, R, DP	C	Z=CMPLX(X,Y)
ICHAR	Converts to integer from character - normally the ASCII value.	1	CHAR	I	I=ICHAR(C)
CHAR	Converts to character from integer, normally the ASCII value.	1	I	CHAR	C=CHAR(I)
AINT	Truncates	1	R, DP	As argument	A=AINT(R)
ANINT	Rounds real and double precision. Yields a real or double precison answer.	1	R, DP	As argument	A=ANINT(R)
NINT	Yields nearest integer	1	R, DP	I	I=NINT(R)
ABS					

Name	Description	Arguments No.	Arguments Type	Result Type	Example
	Yields the absolute value	1	I, R, DP, C	As argument, except complex argument gives real result.	A=ABS(B)
MOD	Returns the remainder when first argument divided by second	1	I, R, DP	As arguments	A=MOD(B,C)
SIGN	Transfer of sign, abs(A1) if A2>=0, –abs(A1) if A2<0	2	I, R, DP	As arguments	A=SIGN(A1,A2)
DIM	Returns first argument minus minimum of the two arguments. A1-MIN(A1,A2)	2	I, R, DP	As arguments	A=DIM(A1,A2)
DPROD	Double precision product of two reals	2	R	DP	D=DPROD(R1,R2)
MAX	Chooses the largest value	[a]	I, R, DP	As arguments	A=MAX(A1,A2,A3)
MIN	Chooses the smallest value	[a]	I, R, DP	As arguments	B=MIN(B1,B2,B3)
LEN	Length of a character entity	1	CHAR	I	L=LEN(C)
INDEX	Locates one substring in another, i.e. returns position of substring C2 in character expression C1	2	CHAR	I	I=INDEX(C1,C2)

Name	Description	Arguments No.	Type	Result Type	Example
AIMAG	Imaginary part of complex argument	1	C	R	Y=AIMAG(Z)
CONJG	Conjugate of a complex argument	1	C	C	Z2=CONJG(Z1)
SQRT	Square root	1	R, DP, C	As argument	X=SQRT(Y)
EXP	Exponential, e^x	1	R, DP, C	As argument	Y=EXP(X)
LOG	Natural logarithm, $\log_e x$	1	R, DP, C	As argument	Y=LOG(X)
LOG10	Common logarithm, $\log_{10} x$	1	R, DP	As argument	Y=LOG10(X)
SIN	Sine	1	R, DP, C	As argument	Y=SIN(X)
COS	Cosine	1	R, DP, C	As argument	Y=COS(X)
TAN	Tangent	1	R, DP	As argument	Y=TAN(X)
ASIN	Arcsine	1	R, DP	As argument	Y=ASIN(X)
ACOS	Arccosine	1	R, DP	As argument	Y=ACOS(X)
ATAN	Arctangent	1	R, DP	As argument	Y=ATAN(X)
ATAN2	Arctangent of A1/A2	2	R, DP	As arguments	A=ATAN2(A1,A2)
SINH	Hyperbolic sine	1	R, DP	As argument	Y=SINH(X)

Name	Description	Arguments No.	Type	Result Type	Example
COSH	Hyperbolic cosine	1	R, DP	As argument	Y=COSH(X)
TANH	Hyperbolic tangent	1	R, DP	As argument	Y=TANH(X)
LGE	Lexically greater than or equal	2	CHAR	L	L=LGE(A,B)
LGT	Lexically greater than	2	CHAR	L	L=LGT(A,B)
LLE	Lexically less than or equal	2	CHAR	L	L=LLE(A,B)
LLT	Lexically less than	2	CHAR	L	L=LLT(A,B)

Notes

For argument type

I=Integer

R=Real

C=Complex

DP=Double Precision

CHAR=Character

L=Logical

[a] Minimum of 2

All angles are expressed in radians

All arguments to an intrinsic function reference must be of the same type