Essentials of FORTRAN 77

John Shelley

Essentials of FORTRAN 77

John Shelley
Imperial College Computer Centre, London

John Wiley & Sons
Chichester · New York · Brisbane · Toronto · Singapore

Library of Congress Cataloging in Publication Data:

Shelley, John.
Essentials of Fortran 77.
Includes index.
1. FORTRAN (Computer program language) I. Title.
II. Title: Essentials of Fortran seventy-seven.
QA76.73.F25S447 1984 001.64'24 84-7452
ISBN 0 471 90502 X

British Library Cataloguing in Publication Data:

Shelley, John, *1940–*
Essentials of Fortran 77.
1. FORTRAN (Computer program language)
I. Title
001.64'24 QA76.73.F25

ISBN 0 471 90502 X

Printed in Great Britain

Dedication: To My Mother

Preface

This text introduces the principal features of programming as well as essential features of the Fortran 77 computer programming language. It is intended for those with little or no previous computing or programming experience, and for those who wish to convert to Fortran 77 from some other language.

After many years of teaching programming to many thousands of students it is clear that the skill of programming is learned only by practice, in the way that the theory of music must be put into practice on some instrument. For this reason, this text has been designed to accompany laboratory sessions during which the program exercises at the end of the chapters can be written and executed. For those interested, Appendix 5 provides a timetable and, Appendix 6, a set of program exercises for an intensive one-week practical programming course held at Imperial College, London.

The text does not attempt to orient the use of Fortran to any particular discipline or group of students. Instead, it sets out to illustrate the *tools* of Fortran 77 and how these are used. It is left to the individual to apply this knowledge to his or her own particular subject, in much the same way that an apprentice carpenter is taught to understand how and when the tools of his trade can be applied.

Chapters 1 and 2 are intended to introduce the principles of problem solving with the aid of a computer. For the beginner, this is not an easy task to come to terms with since it requires an entirely new approach to problem solving. These two chapters should be read carefully. The rest of the text concentrates upon essential features of Fortran 77.

Fortran 77, however, is a large and complex language. It is easy to confuse novices by presenting at one time *all* the variations of a particular programming feature. Hence, care is taken to present in some cases just part of the total picture. Once students have gained confidence with this part, they can move onto to the more detailed parts. In order to achieve this, the material has been organised at various levels or *cycles* as I have called them. By following all the material at cycle 1 and then returning to a study of all the material at cycle 2 and, later, cycle 3, students will be able to absorb relevant aspects best suited to their own level of experience. The exercises and tests at the end of Chapters are also arranged to follow this cyclic method. As a colleague of mine pointed out, this is the first 'circular' text he had come across. However, readers with some previous programming experience and,

sufficient confidence, may ignore the cycles altogether.

In most practical programming courses, for which this text is intended, there is time only to present the essential features of Fortran 77. It is better for the student to become confident in the use of this *working set* and, later, to discover the rest, than to flood the mind with so much that very little can be assimilated. Thus, we do not claim to teach all of Fortran 77 preferring, instead, to teach what is most needed. The purpose of the final chapter is to point towards other possible 'useful' features which students can study at a future time.

John Shelley M.Phil., DIC, MBCS.
January, 1984.

CYCLE CHART

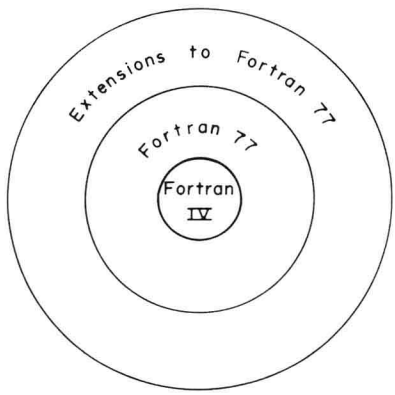
Text	Appendix 1	Chapter 1	2	3	4	5	6	7	8	9	10	11	12	13	App 2	App 3	App 4	App 5	App 6
Cycle 1	§	§	§	§	§	§	§	§		§	§	§							
2					§	§	§		§	§	§								
3							§						§	§	§	§	§		
4	On your own with Computer Centre's reference manual																		

Note to the Reader

Fortran, the first of the so-called 'high-level-languages' was developed in 1956 for use on an IBM 704 computer. During the middle 1960's, it became widely used on a number of machines resulting in a variety of 'dialects'. By 1962, the American Standards Institute (later to become the American National Standards Institute - ANSI) set up a working party to produce a specification for the language. In March, 1966, two versions were approved - Fortran II and Fortran IV. Their popularity grew to such an extent that most computer manufacturers had to ensure that their dialect of Fortran conformed to these standards.

Fortran IV (sometimes referred to as Fortran 66 - the year of its final approval), the more popular of the two standards, became the standard Fortran language. A decade later another standard Fortran language was specified, Fortran 77. This accepts most of Fortran IV but has certain additional features such as the IF-THEN-ELSE construction, CHARACTER type, enhancements to the DO loop counter and the index of subscripted variables.

In the diagram below, Fortran IV is seen to be within the Fortran 77 language indicating that what is in the Fortran IV specification is also accepted in the Fortran 77 standard. But there is also an area of *extensions*. These are features over and above the standards laid down by the 1977 Fortran Committee and are defined by individual computer manufacturers and installations. They apply in particular to file manipulation, character encoding and input and output facilities. The way to discover these extensions is to consult the reference manual supplied by the manufacturer or installation.



In this text, we shall concentrate on standard Fortran 77 and not the particular extensions permitted by individual computer systems. Since some readers may well have to refer to programs written in Fortran IV, we take the precaution of stating where appropriate the differences between Fortran IV and Fortran 77.

Your Guarantee: Each program in this text, except for those in the Exercises which contain deliberate errors, has been executed on a computer (a CDC Cyber 174) to ensure that it is correct.

The Lecturer's Progress

As lecturers we need to undergo three phases. In the first phase we tend to teach all we know about a subject. During the second phase, we teach all we did not know the first time round. But in the third and final phase, we have, at last, learned to teach what students need to know.

Contents

Chapter 1: Introduction to Programming

☞ *Cycle 1:*

First exposure to programming a computer can be a somewhat daunting experience, especially if you have no understanding of what a computer is or what it can do and are totally confused by all the jargon. If you are new or fairly new to computing, you are strongly advised to read Appendix 1 (The Computing Process) before proceeding any further.

This Appendix will explain briefly what computers are, why people want to use them and how human beings communicate with computers. It will introduce the essential and common jargon terms; and, most important, for the aspiring programmer, it will outline the basic operations performed by any computer.

A Fundamental Point

No matter how much a computer costs, be it £5 or £5 million, no matter how large or small, a computer can perform but *four* basic operations:

- input & output operations;
- arithmetic operations;
- decisions (via comparison & logic operations);
- movement (and structuring) of data operations inside the CPU.

The skill (some prefer to call it an art) of programming lies in the ability of the programmer to break down some overall problem, such as space exploration, traffic control, retail stock control, into an interplay between these four elementary operations. No matter how sophisticated a programming language is claimed to be any instruction in that language will be seen to be one or a combination of these four operations.

Because of the basic nature of the type of operations performed by computers, a program has to be very detailed, much more detailed than we are used to when conversing with each other. Programming then can be an arduous, lengthy and expensive business.

A Moral

Computers are of value as problem solvers only when the human effort involved in writing the program is *less than* the effort involved using some other method. Thus, if you want to add up some numbers, reach for a pocket calculator.

However, in order to demonstrate the techniques of programming, we shall have to use some simple examples, frequently of an arithmetical nature. This is forced upon us at the teaching stage. It is left to you to remember that we are not necessarily advocating the use of computers in such simple cases.

Our First Problem

Suppose we want a human colleague to convert temperatures from Fahrenheit to Centigrade and vice versa. This might be a problem set for school children. We would have to provide two elements:

- an instruction (convert from Fahrenheit to Centigrade and vice versa)
- a list of temperatures.

List of Temperatures	*Instruction*
Fah: 21° *Cent: -32°* *Fah: -102.5°* *Fah: 356°* *Cent: 235.5°*	*Convert Fahrenheit to Centigrade and vice versa.*

Figure 1.1

If we want to use a computer to do the same thing, we must also supply the same two elements. To use computer jargon, the instruction is called a *program* and the temperature values called *data values* or just *data* for short. However, the program (the collection of instructions to solve a given problem) could well involve some 7 or more instructions and not just the one when involving a human being. It is the detailed level of program instructions which distinguishes between the human being and the computer as problem solvers.

Where To Begin

How do we begin to convert a problem into a format which can be readily understood by a computer? After we have understood the problem itself, there are three points to bear in mind.

Know Thy Data

This is most important. For many programmers, this must frequently be the actual starting point of the entire programming process. Unless we know *exactly* the nature of our data, we cannot solve the problem. We shall have need to return to this point from time to time.

Furthermore, it is necessary to know how the data is to be organised for input into the computer. Sometimes, programmers can decide this for themselves. But with most serious programs, this choice will not be there. Someone else will have already decided upon the format of the data. Therefore, as a programmer, you will need to find out how the data has been arranged so that your program can call it into the computer memory.

Data is in the External World

A program is written assuming that it is inside the computer memory whilst the data upon which it will work lies outside. The program, via an *input* operation will have to call the data from the outside world into central memory (CM). Figure 1.2 shows that data may come from a card reader (i.e. data is punched as holes on a card), typed in at a keyboard, or, indeed, stored on some magnetic media. At the simplest level, the computing process is an 'input of data' - 'process data' - 'output results' procedure (I-P-O).

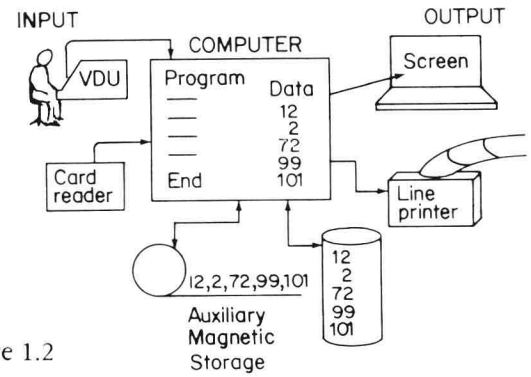


Figure 1.2

Programs are General

A program has a *general* air about it. It does not make sense to write a program to add up two specific numbers (e.g. 4 and 5). Should I want two different numbers to be added up, e.g. 12 and 23, I would have to write another program. Thus, when I do write the program, I make sure that it is general enough so that it will add up *any* two numbers.

In our case, we shall try to make our program so general that it will convert *any* list of temperatures which we care to provide. This is fundamental to programming. No one writes a payroll program or a graphics program to perform just one payroll or to design just one car. Payroll programs are written to perform the same function every week; design programs are written to function for any possible car design. Bearing these three points in mind, how do we now begin with our problem?

Temperature Conversions

First, what do we know about our data?

- the data is in two parts, one part indicates the type of degree, the other is the temperature itself; e.g: Fahrenheit, 23; Centigrade, -102.
- the number of conversions to perform is unknown.

We also need to know how the pair of values are to be entered into the computer. In our case, and to make matters simple at the start, the data is to be entered at a

keyboard as a pair, the first being the 'type indicator', the second being the 'temperature'. The types of temperatures may be entered in any sequence. This may seem unnecessarily detailed, but you will soon discover that without such detail you cannot write the program.

Thus, everytime a pair is entered, the computer via our program will need to determine the type of degree, convert to the other form, and then request another pair. This procedure will continue until there are no more values to enter.

We must now provide the computer with a sequence of steps which it will follow one after the other until it is told to stop doing any more, i.e. to stop the execution of the program. The idea in programming is for the computer to begin at step one, (the first instruction), and to proceed in strict sequence through the rest of the instructions. There is nothing unusual about this. We do the same when following a cooking recipe, a knitting pattern, the instructions for constructing a piece of furniture, etc.

We need to create or design a sequence of steps which as a total set will solve the given problem. This is more formally called an *algorithm*. Since we have no idea of how to write program instructions, we shall present a series of steps using concise English phrases which can later be converted (coded) into programming instructions.

So, what should our first 'instruction' be? Clearly, since the program is inside the CM and the values are lying outside waiting to be read in, we shall need to instruct the computer to perform an input (reading in) operation to bring in the first pair. This is step 1 in figure 1.3. The next step is to determine which type of degree has been entered. If it is Fahrenheit, then the temperature needs to be converted to Centigrade and the result printed out; otherwise, as step 3, if the type is Centigrade, then convert to Fahrenheit and print result. Having completed the conversion on the first pair, the program must now instruct the computer to read in a second pair, perform the conversion, print result; read in pair 3; etc.

step 1: read in *FIRST PAIR* of **TYPE & TEMP**

step 2: if **TYPE** is Fahrenheit, convert **TEMP** to Centigrade, print result

step 3: otherwise, convert **TEMP** to Fahrenheit and print result

step 4: read in *SECOND PAIR* of **TYPE & TEMP**

step 5: if **TYPE** is Fahrenheit, convert **TEMP** to Centigrade and print result

step 6: otherwise, convert **TEMP** to Fahrenheit and print result

step 7: read in *THIRD PAIR* of **TYPE & TEMP** etc...

Figure 1.3

However, if we have 20 or 20 000 pairs in the set, this procedure becomes not only tedious, but so time-consuming to write out that it would be quicker to do the whole thing by hand. Furthermore, we should have to know the exact number of pairs to read in. This is seldom the case for the majority of programs.

A Basic Pattern

One point emerges from our first simple approach to the solution. There is a basic set of actions which is similar for each pair. This basic pattern is enclosed in the brackets, and consists of reading in *another* pair, determining the type and then converting the temperature to the opposite. As you gain experience, you will discover that unless a basic pattern begins to emerge quite quickly, it could be that the computer is *not* the best tool to use, and you may have to consider some alternative.

Provided that we are not specific in reading in a given pair (the first, the second, etc), we can reduce the number of steps as shown in Figure 1.4. By reading in *another pair*, we can repeat the steps enclosed in the bracket. This is one of the most significant features of programming, the ability to repeat the same instructions many times. Note well, that the program now has a *general air* about it.

read in **ANOTHER PAIR**

if **TYPE** is Fahrenheit, then convert **TEMP** to Centigrade and print result

otherwise, convert **TEMP** to Fahrenheit and print result

Repeat until no more pairs

Stop execution of program.

Figure 1.4

Note that although the basic pattern in the bracket is a repeated sequence, it is *not* an exact repetition. Why not?

In fact, it is a *similar* sequence but working on different data values each time. This means that we need to pay great attention to the *names* used in instructions. Chapter 3 will discuss this point in detail.

Before going any further, try a few simple exercises in order to consolidate the features presented in this chapter. It will make you begin to think about a problem in the way that a programmer has to when first presented with a problem for computer solution. In the exercises, use the method discussed in this chapter. Remember the four basic operations which computers perform and use concise English phrases to describe the separate steps in the solutions. If you find it strange and difficult to think out a problem in such detail do not be surprised or put off.

Experience gained from practice will make it easier.

Exercises

Cycle 1:

1. Find the sum of a set of ten positive numbers.
2. Extend the above to compute the average of the sum.

3. Find the average speed and cost of petrol for a car trip involving the 3 journeys below:

A ⇒B: 50 miles in 2 ½ hrs
B ⇒C: 193 miles in 3 hrs
C ⇒A: 10 miles in 15 mins.

Assume the car does 40 miles to the gallon @£1.83 per gallon.

where: average speed = distance ÷time.

Test

Cycle 1:

1. Fill in the missing line:

Know Thy Data
.....
Programs are General

2. Explain what is meant by programs being *general*.

3. List the four basic operations a computer can only perform.