Programming in FORTRAN

Third Edition

V. J. Calderbank

Chapman and Hall Computing

Programming in FORTRAN

Third edition

V. J. Calderbank
Information Technology Division, UKAEA
Culham Laboratory

London New York
CHAPMAN AND HALL

First published in 1969 by
Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE
Published in the USA by
Chapman and Hall
29 West 35th Street, New York NY 10001
Second edition 1983
Third edition 1989

Typeset in 10/12 Times by
Colset Private Ltd, Singapore
Printed in Great Britain by
T. J. Press Ltd, Padstow, Cornwall

ISBN 0 412 30500 3 (hardback)
0 412 30510 0 (paperback)

British Library Cataloguing in Publication Data

Calderbank, Valerie J. (Valerie Joyce), *1944–*
Programming for FORTRAN. — 3rd ed.
1. Computer systems. Programming languages
: FORTRAN 77 language
I. Title II. Calderbank, Valerie J.
(Valerie Joyce), *1944–*. Course on
programming on FORTRAN
005.13'3

ISBN 0-412-30500-3
ISBN 0-412-30510-0 Pbk

Library of Congress Cataloging in Publication Data

Calderbank, Valerie Joyce.
Programming in FORTRAN / V.J. Calderbank. — 3rd ed.
p. cm.
Rev. ed. of: A course on programming in FORTRAN. 2nd ed., rev. to incorporate FORTRAN 77. c1983.
Includes index.
ISBN 0-412-30500-3. ISBN 0-412-30510-0 (pbk.)
1. FORTRAN (Computer program language) I. Calderbank, Valerie Joyce. Course on programming in FORTRAN. II. Title.
QA76.73.F25C35 1989
005.13'3—dc19 88-25964
CIP

Programming in FORTRAN

CHAPMAN AND HALL COMPUTING SERIES

COMPUTER OPERATING SYSTEMS
For micros, minis and mainframes
2nd edition
David Barron

MICROCOMPUTER GRAPICS
Michael Batty

THE PICK OPERATING SYSTEM
Malcolm Bull

PROGRAMMING IN FORTRAN
3rd edition
V.J. Calderbank

EXPERT SYSTEMS
Principles and case studies
2nd edition
Edited by Richard Forsyth

MACHINE LEARNING
Principles and techniques
Edited by Richard Forsyth

EXPERT SYSTEMS
Knowledge, uncertainty and decision
Ian Graham and Peter Llewelyn Jones

COMPUTER GRAPHICS AND APPLICATIONS
Dennis Harris

ARTIFICIAL INTELLIGENCE AND HUMAN LEARNING
Intelligent computer-aided instruction
Edited by John Self

ARTIFICIAL INTELLIGENCE
Principles and applications
Edited by Masoud Yazdani

Preface

During the eighteen years since this book was first published the success of FORTRAN as a programming language has continued unabated. The language was first standardized in 1966 by the American National Standards Institute (ANSI) and this definition of the language was, by and large, the subject of the first edition of the book. In 1977 the ANSI committee published a revised standard definition of the language which became known as ANSI 77 FORTRAN or simply FORTRAN 77 (see ANSI X3.9, 1978 for a description of this). The second edition of this book was published at a time when there was widespread interest in this language, but compilers for it were not universally available. This edition therefore acted as a transition book and described both FORTRAN 66 and FORTRAN 77 side by side. There is now no doubt that almost all FORTRAN programmers today program in FORTRAN 77, and interest in the old standard has largely died out. In fact, the FORTRAN community is now looking forward to the finalization of the ANSI standard for FORTRAN 8x which should appear in the next year. Many new facilities are being added to the 8x standard and many features of the old FORTRAN will be deprecated or become obsolete.

It is with all this in mind that this third edition has been written. First, it seeks to teach FORTRAN 77 from scratch without reference to FORTRAN 66, and therefore topics are introduced in a somewhat different order from the way they were handled in the second edition. Second, it tries to push into the background features of FORTRAN 77 which are undesirable, partly with the intention of encouraging better programming techniques and partly with an eye to the new 8x standard. Third, it tries to encourage the use of structured programming techniques in the hope of fostering improved software standards, particularly among the increasing army of D.I.Y. programmers who are not computer professionals, but who nevertheless spend many man-days programming personal computers or scientific workstations. It is for my readers to determine whether I have succeeded in my aims.

I have many people to thank for help in the production of various editions of this book. My special thanks must always go to Professor E. J. Burge, Head of the Physics Department of Chelsea College, London, for giving me the courage to write the first edition. Special thanks also to my family for enabling me to spend the hours which are inevitably necessary in the

production of a book of this kind. Thanks, of course, must go to the numerous scientific colleagues, university lecturers, students and critics who have contributed to what I hope is a FORTRAN text which has gradually improved over the years. Last, but not least, I must thank my publishers (Chapman and Hall) for their continuing confidence in me and for the many hours of work which they have put into all three editions.

V. J. Calderbank
Information Technology Division
Culham Laboratory
1988

Contents

1

Fundamentals

1.1 Basic computer concepts

The purpose of this book is to teach the reader to program a computer in the FORTRAN programming language. It is not intended to discuss the way in which a modern computer is constructed or operated. However, some knowledge of the way a computer works is required before any attempt can be made to program it, and therefore a brief introduction is given here.

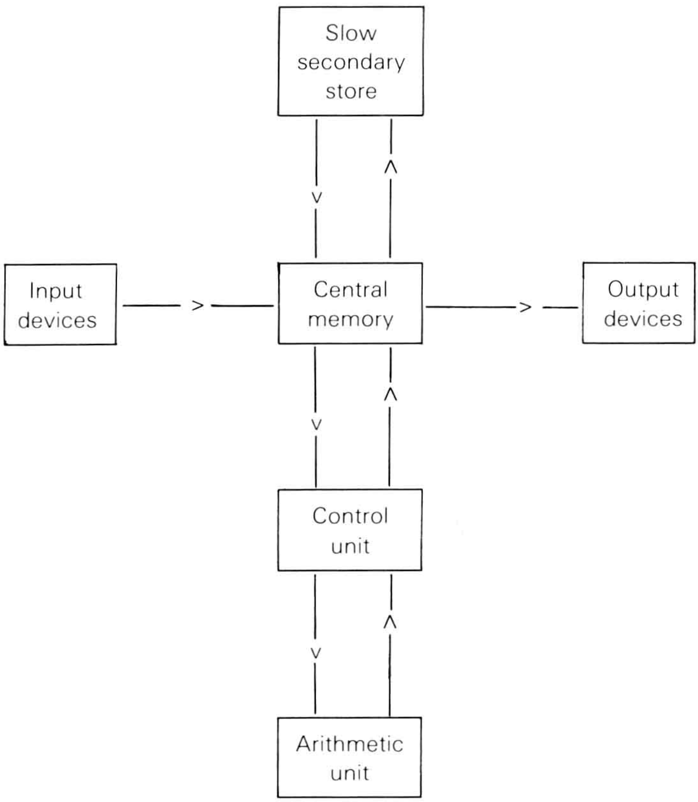

FIG. 1.1 A schematic diagram of a typical computer system

Computer architecture can vary greatly in detailed design but the basic principle of most systems may be represented by the diagram in Fig. 1.1. Instructions or data are entered into the computer via an *input device* and stored in the central *memory*. The basic unit of memory is the *binary digit* (or bit). For convenience, bits are grouped together in larger units called *bytes* (typically 8 bits) and *words* (which are different sizes on different computers but may typically be 8, 16, 32, 48 or 64 bits). Note that a computer word is a collection of binary digits and as such is very different from a natural language word which is a collection of characters. Natural language words may be stored in computer words, however, using a numerical code to represent each character.

The position of a word or byte in computer memory is known as its *address*. To clarify this with an everyday analogy, think of a computer's memory as a chest of drawers where each drawer is divided into, say, 16 compartments across its width. This is shown in the diagram of Fig. 1.2. Imagine that each drawer is equivalent to a computer word and each compartment is equivalent to a computer bit; then this chest of drawers represents a 16 bit word memory (each of which may contain two 8 bit bytes).

If we number the drawers from 0 to 5 starting from the top, then the third drawer down is at address 2, the fourth at address 3 and so on (this numbering is chosen because computer memory is often addressed from 0 upwards). Within the drawer, the compartments themselves are numbered from 0 to 15, starting from the right. This is equivalent to bit 0, bit 1 and so on in the computer word.

Let us suppose that any compartment in this drawer is allowed to contain either one item or nothing at all. Then this is analogous to a computer bit

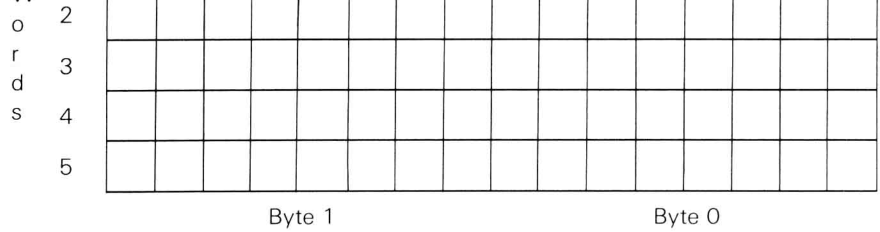

FIG. 1.2 A schematic representation of computer memory

which can hold the value 1 or 0 (often represented electronically by a current being on or off).

All information, whether instructions or data, is held in the computer's memory in this way. Groups of binary digits can represent larger numbers, or characters held in a coded numerical form. For example, one bit can only hold the value 0 or 1, but two bits can represent 00, 01, 10 or 11 (this will be described more fully when internal representation is discussed later).

Once data has been entered and stored in the computer's memory, it may be required to perform calculations on it. To do this, the data must be moved from memory locations, along a data highway or *high-speed bus* to the *arithmetic unit* which contains one or more accumulators or high-speed working *registers*. Very fast computations may be carried out in these registers and the results returned to memory along the bus. Computers vary in the number of bits of information that can be carried along the bus in one operation and in the number that can be held in the registers, e.g. there may be an 8 bit bus and 16 bit registers; this basic architecture obviously affects the speed of the machine.

Instructions to move data and perform calculations on it are held in the memory in the form of a *program* and it is the *control unit* which controls and interprets these instructions. It is able to access instructions stored sequentially in the memory, decode them and initiate the appropriate action. The arithmetic unit, control unit, registers and central memory together form the *central processor unit (CPU)*. To this are connected *input/output* devices such as *visual display units*, *teletypes and lineprinters*, and storage devices (such as magnetic tapes, discs and drums) known as *secondary store* or backing store. All of these devices are collectively known as *peripherals*.

A typical modern scientific computer installation will provide a variety of peripherals such as on-line terminals (teletypes and visual display units) for both input and output, magnetic tapes, discs, drums, cassettes and cartridges for storing programs and data, and output devices such as lineprinters and graph plotters.

1.2 Algorithms

The CPU of any computer contains logic circuits which can themselves perform simple basic instructions such as addition, subtraction, multiplication, division and so on. Thus a circuit may take as its input two 16 bit numbers, say, and produce, by means of logic gates, one output which is the sum of these two. More complex operations are performed by breaking them down into logical sequences of these basic operations. So just as it is possible to build a palace from a few basic ingredients such as bricks and mortar, so it is possible to solve enormously complex problems using the basic instruction set of a computer. But just as the architect has to provide instructions and blueprints from which the palace can be built, so must the programmer provide instructions and diagrams from which a program can be written to solve a particular problem. In computing and mathematics, a prescription for solving a problem is generally called an *algorithm* whether or not it ultimately will become a computer program.

There are many everyday examples of algorithms; a recipe for making a cake is a set of instructions which when obeyed in sequence result in a cake. Thus the recipe is an algorithm for performing this task. Another example may be a list of directions to reach your home from your place of work, e.g. turn left at the main exit, drive straight on for 2 miles, at the next T junction turn right and so on. You may give these written instructions to any number of colleagues, and if they are clear and unambiguous then they should all ultimately arrive at your home. On the other hand, if they are ambiguous or incorrect then your colleagues will lose their way or perhaps spend hours driving around in circles. So it is with algorithms for computer programs. A computer will obey, in a moronic way, the exact instructions that it is given. This may result in the wrong answer or in the CPU looping for perhaps hours.

Let us consider algorithms in a little more detail by returning to the analogy of a chest of drawers introduced in the previous section. Suppose that the first drawer contains one knife, the second drawer contains one fork and the third drawer contains one spoon in each compartment, then the instructions to lay one place setting on a table may be as follows:

Open drawer 0
Take a knife from compartment 0
Close drawer 0
Open drawer 1
Take a fork from compartment 0
Close drawer 1
Open drawer 2
Take a spoon from compartment 0
Close drawer 2
Go to the table
Lay one knife, one fork and one spoon on the table

This algorithm may be repeated three more times to lay four place settings but each time taking the knives, forks and spoons from a different compartment, i.e. 1, 2 and 3.

So this is an algorithm, but not one which can be directly converted into a computer program. Consider now a more numerical algorithm to add two numbers and print the result. This might read as follows:

Read the first number
Read the second number
Add the two numbers together
Print the result

This is obviously a simple but limited task that a computer could perform and we shall see later how this becomes a FORTRAN program. It consists of a sequence of four basic instructions and that is all. A slightly more powerful algorithm might sum the squares of N numbers and print the result as follows:

Read N
Set the 'sum so far' to zero
Repeat the following N times:
    Read a number
    Square it
    Add the result to the 'sum so far'

This algorithm spells out, in an English-like language, the primitive steps which have to be performed to sum the squares of N numbers. It consists of two basic constructs – elementary operations or commands which must be obeyed in sequence (e.g. Square it, Add the result to the 'sum so far'), and repetitions which require a group of instructions to be performed repeatedly, usually not for evermore, but until some condition becomes true (in this case, until they have been obeyed N times).

It can be seen from this that algorithms can be written in a type of English regardless of whether they are to be subsequently written as actual programs for a machine to obey or not. Here we must make one of the first important statements about how to write a computer program. Spend a long time designing your algorithm before you make any attempt to program it. There is a widely held view that the sooner you actually start programming, the longer the program will take to develop.

Many programs do not work because the algorithms do not work in the first place, just as your colleagues will get lost if you give them wrong directions. Only when you are sure that your design is right should you tackle the separate problem of coding this in a computer language for input to a computer.

So, to summarize, a computer is able to produce a solution to a particular problem only if it is presented with a series of simple instructions which will, when obeyed in a specific order, produce the desired result. This sequence of instructions is referred to as a program; programs are collectively termed *computer software* in contrast to the actual physical devices which collectively form the *hardware*. The instructions which form a program are loaded into the computer's memory in an encoded form and the control unit works sequentially through the instructions, decoding and obeying them. In so doing, it may use data stored at other locations in memory.

1.3 Structured design

The algorithms in the preceding section illustrate the use of two important elements in program design – sequences of elementary operations and repetitions. A third important construct for writing algorithms is the selection. Suppose we wish to calculate the income tax payable on a number of salaries in the range £1 to £20 000. Suppose tax is to be calculated at a rate of 25% on the first £750, 30% on the next £5000 and 45% on the remainder. Let us suppose that the calculation is to end when a salary is found which is not in the above range.

We shall approach the design of this algorithm using *top–down design*. That is, first of all we state the overall aim of the program which may be represented by:

Calculate and print income taxes

We now break this down a bit further by defining the process 'calculate and print income taxes'; this is a repetition of 'calculate tax' and 'print tax' which in turn are defined as follows:

```
Read salary
If the salary is between £1 and £20 000 then
     Calculate tax
     Print tax
else
     ***Error – invalid salary
end if
```

The process 'calculate tax' may be further broken down into its basic tasks as follows:

```
Calculate tax at 25% on the first £750
If the salary is greater than £750 then
     Calculate tax at 30% on the next £5000
     If the salary is greater than £5750 then
          Calculate tax on the remainder at 45%
     end if
end if
```

This example provides several illustrations of the selection which is of the form 'If this is true then do the following' and may or may not have an alternative clause 'else do the following'. The end of each 'if' clause is marked by a corresponding 'end if' for clarity.

These then are the basic logical elements which may be combined to form an algorithm – *sequences*, *selections* and *repetitions* (also called *loops* or *iterations*). Using these and top–down analysis, large and complicated programs may be broken down into small manageable tasks, each of which