Ian Chivers · Jane Sleightholme

Introduction to Programming with Fortran

With Coverage of Fortran 90, 95, 2003, 2008 and 77

Second Edition

Springer

Introduction to Programming with Fortran

Ian Chivers • Jane Sleightholme

Introduction to Programming with Fortran

With Coverage of Fortran 90, 95, 2003, 2008 and 77

Springer

Ian Chivers
Rhymney Consulting
UK

Jane Sleightholme
Fortranplus
UK

ISBN 978-0-85729-232-2 e-ISBN 978-0-85729-233-9
DOI 10.1007/978-0-85729-233-9
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2011941580

© Springer-Verlag London Limited 2012
Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.
The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.
The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Acknowledgement

The material in the book has evolved firstly from our combined experience of working in Computing Services within the University of London at

- King's College, IDC (1986–2002) and JS (1985–2008)
- Chelsea College, JS (1978–1985)
- Imperial College, IDC (1978–1986)

in the teaching, advice and support of Fortran and related areas, and secondly in the provision of commercial training courses. The following are some of the organisations we've provided training for:

- AWE, Aldermaston.
- Centre for Ecology and Hydrology, Wallingford.
- DTU – Danish Technical University.
- Environment Agency, Worthing.
- JET – Joint European Torus.
- The Met Office, Bracknell and Exeter.
- Natural Resources Canada, Ottowa.
- QinetiQ, Farnborough.
- Rolls Royce, Derby.
- SHMU, Slovak Hydrometeorological Institute, Bratislava, Slovakia.
- University of Ulster, Jordanstown, Northern Ireland.
- Veritas DGC Ltd., Crawley.
- Westland Helicopters, Yeovil.

The examples in the book are based on what will work with compilers that support the Fortran 2008 standard.

Thanks are due to:

- The staff and students at King's College, Chelsea College and Imperial College.
- The people who have attended the commercial courses. Its been great fun teaching you and things have been very lively at times.

- The people on the Fortran 90 list and comp.lang.fortran. Access to the expertise of several hundred people involved in the use and development of Fortran on a daily basis across a wide range of disciplines is inestimable.
- The people at NAG for the provision of beta test versions of their Fortran compilers.
- The people at Intel for the provision of beta test versions of their Fortran compilers.
- The staff and facilities at PTR Associates. It is a pleasure training there.
- Helmut Michels at the Max-Planck-Institut for permission to use the dislin library.
- The patience of our families during the time required to develop the courses upon which this book is based and whilst preparing the camera-ready copy.
- Finally Helen Desmond and Beverley Ford at Springer for their enthusiasm and encouragement!

Our Fortran home page is:

- http://www.fortranplus.co.uk/

All of the program examples can be found there.
If you would like to contact us our email addresses are:
Ian D Chivers: ian@rhymneyconsulting.co.uk
Jane Sleightholme: jane@fortranplus.co.uk

Contents

Chapter 1
Overview

I don't know what the language of the year 2000 will look like, but it will be called Fortran.

C.A.R. Hoare

1.1 Introduction

The book aims to provide coverage of a reasonable working subset of the Fortran programming language. The subset chosen should enable you to solve quite a wide range of frequently occurring problems.

This book has been written for both complete beginners with little or no programming background and experienced Fortran programmers who want to update their skills and move to a modern version of the language.

Chapters 2 and 3 provide a coverage of problem solving and the history and development of programming languages. Chapter 2 is essential for the beginner as the concepts introduced there are used and expanded on throughout the rest of the book. Chapter 3 should be read at some point but can be omitted initially. Programming languages evolve and some understanding of where Fortran has come from and where it is going will prove valuable in the longer term.

- Chapter 2 looks at problem solving in some depth, and there is a coverage of the way we define problems, the role of algorithms, the use of both top-down and bottom-up methods, and the requirement for formal systems analysis and design for more complex problems.
- Chapter 3 looks at the history and development of programming languages. This is essential as Fortran has evolved considerably from its origins in the mid-1950s, through the first standard in 1966, the Fortran 77 standard, the Fortran 90 standard, the Fortran 95 standard, TR 15580 and TR 15581, Fortran 2003 and Fortran 2008. It helps to put many of the current and proposed features of Fortran into

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_1, © Springer-Verlag London Limited 2012

context. Languages covered include Cobol, Algol, Lisp, Snobol, PL/1, Algol 68, Simula, Pascal, APL, Basic, C, Ada, Modula, Modula 2, Logo, Prolog, SQL, ICON, Oberon, Oberon 2, Smalltalk, C++, C# and Java.

Chapters 4 through 8 cover the major features provided in Fortran for numeric programming in the first instance and for general purpose programming in the second. Each chapter has a set of problems. It is essential that a reasonable range of problems are attempted and completed, as it is impossible to learn any language without practice.

- Chapter 4 provides an introduction to programming with some simple Fortran examples. For people with a knowledge of programming this chapter can be covered fairly quickly.
- Chapter 5 looks at arithmetic in some depth, with a coverage of the various numeric data types, expressions and assignment of scalar variables. There is also a thorough coverage of the facilities provided in Fortran to help write programs that work on different hardware platforms.
- Chapter 6 is an introduction to arrays and do loops. The chapter starts with some examples of tabular structures that one should be familiar with. There is then an examination of what concepts we need in a programming language to support manipulation of tabular data.
- Chapter 7 takes the ideas introduced in Chap. 6 and extends them to higher-dimensioned arrays, additional forms of the dimension attribute and corresponding form of the do loop, and the use of looping for the control of repetition and manipulation of tabular information without the use of arrays.
- Chapter 8 looks at more of the facilities offered for the manipulation of whole arrays and array sections, ways in which we can initialise arrays using constructors, look more formally at the concepts we need to be able to accurately describe and understand arrays, and finally look at the differences between the way Fortran allows us to use arrays and the mathematical rules governing matrices.

Chapters 9, 10 and 11 look at input and output (I/O) and file handling in Fortran. An understanding of I/O is necessary for the development of so-called production, non interactive programs. These are essentially fully developed programs that are used repeatedly with a variety of data inputs and results.

- Chapter 9 looks at output of results and how to generate something that is more comprehensible and easy to read than what is available with free format output and also how to write the results to a file rather than the screen.
- Chapter 10 extends the ideas introduced in Chap. 12 on output to cover input of data, or reading data into a program and also considers file I/O.
- Chapter 11 provides a coverage of files.

Chapter 12 introduces the first building block available in Fortran for the construction of programs for the solution of larger, more complex problems. It looks at the functions available in Fortran, the so-called intrinsic functions and procedures (over 100 of them) and covers how you can define and use your own functions.

It is essential to develop an understanding of the functions provided by the language and when it is necessary to write your own.

Chapter 13 introduces more formally the concept of control structures and their role in structured programming. Some of the control structures available in Fortran are introduced in earlier chapters, but there is a summary here of those already covered plus several new ones that complete our coverage of a minimal working set.

Chapters 14 through 16 complete our coverage of the intrinsic facilities in Fortran for data typing.

- Chapter 14 looks at the character data type in Fortran. There is a coverage of I/O again, with the operators available—only one in fact.
- Chapter 15 looks at the last numeric data type in Fortran, the complex data type. This data type is essential to the solution of a small class of problems in mathematics and engineering.
- Chapter 16 looks at the logical data type. The material covered here helps considerably in increasing the power and sophistication of the way we use and construct logical expressions in Fortran. This proves invaluable in the construction and use of logical expressions in control structures.

Chapter 17 introduces derived or user defined types with a small number of examples.

Chapter 18 looks at the dynamic data-structuring facilities now available in Fortran with the addition of pointers. This chapter looks at the basic syntax of pointers. They are used in range of examples in later chapters in the book.

The next two chapters look at the second major building block in Fortran—the subroutine. Chapter 19 provides a gentle introduction to some of the fundamental concepts of subroutine definition and use and Chapter 20 extends these ideas.

Chapter 21 introduces one of modern Fortran's major key features— the module. A Fortran module can be thought of as equivalent to a class in C++, Java and C#. This chapter looks at the basic syntax, with a couple of simple examples.

Chapter 22 looks at simple data structuring in Fortran, as we have now covered modules in a bit more depth.

Chapter 23 looks briefly at operator overloading, first introduced in Fortran 90.

Chapter 24 looks at generic programming.

Chapter 25 has a small set of mathematical examples.

Chapter 26 introduces object oriented programming in Fortran.

Chapters 27 through 30 look at parallel programming in Fortran with coverage of MPI, OpenMP and Coarray Fortran.

Chapter 31 looks at C interoperability.

Chapter 32 looks at IEEE Arithmetic support in Fortran.

Chapter 33 looks at a number of miscellaneous Fortran features.

Chapter 34 looks at converting from Fortran 77 to more modern Fortran.

Some of the chapters have annotated bibliographies. These often have pointers and directions for further reading. The coverage provided cannot be seen in isolation. The concepts introduced are by intention brief, and fuller coverage must be sought where necessary.

There are several appendices:

- Appendix A—This is a glossary which provides coverage of both the new concepts provided by Fortran and a range of computing terms and ideas.
- Appendix B—The ASCII character set.
- Appendix C—Contains a list of some of the more commonly used intrinsic procedures in Fortran and includes an explanation of each procedure with a coverage of the rules and restrictions that apply and examples of use where appropriate.
- Appendix D—Contains the English and Latin text extracts used in one of the problems in the chapter on characters.
- Appendix E—Contains the coded text extract used in one of the problems in Chapter 17.
- Appendix F—Formal syntax
- Appendix G—Sample compiler options

This book is not and cannot possibly be completely self-contained and exhaustive in its coverage of the Fortran language. Our first intention has been to produce a coverage of the features that will get you started with Fortran and enable you to solve a range of problems successfully.

All in all Fortran is an exciting language, and it has caught up with language developments of the last 50 years.

Several Fortran compilers have been used whilst writing this book. These include:

- NAG Fortran Builder 5.1, 5.2, for Windows
- NAG Fortran Compiler 5.1, 5.2, 5.3 for Windows
- NAG Fortran Compiler 5.1, 5.2 for Linux.
- Intel Fortran 11.x, 12.x for Windows.
- Intel Fortran 12.x for Linux.
- gnu gfortran 4.x for Windows.
- gnu gfortran 4.x for Linux.
- g95 for Linux.
- pgi 10.x—Cray Hector service
- Cray 1.0.1—Cray Hector service
- Oracle Solaris Studio 12.0, 12.1, 12.2 for Linux

Our recommendation is that you use at least two compilers in the development of your code. Moving code between compilers and platforms teaches you a lot.

We are the current owners of the Fortran 90 list, and quoting the introduction "This list covers all aspects of Fortran 90 and HPF, the new standard(s) for Fortran. The emphasis should be on the *new* features of Fortran 90. It welcomes contributions from people who write Fortran 90 applications, teach it in courses, want to port programs and use it on (super)computers."

Visit:

- http://www.jiscmail.ac.uk/lists/comp-fortran-90.html

for more information.

Ian Chivers is also Editor of Fortran Forum, the SIGPLAN Special Interest Publication on Fortran, ACM Press. Visit

http://portal.acm.org/citation.cfm?id=J286

for more information.

1.2 Program Examples

All of the program examples are available on line at

`http://www.fortranplus.co.uk/`

1.3 Further Reading

Mastery of any programming language requires working with technical documentation. You will have to refer to or use one or more of the sources below if you want to progress as a Fortran programmer.

1.3.1 The Fortran Standard

The ISO site

http://www.iso.org/iso/search.htm?qt=fortran&sort=rel&type=simple&published=on

has details of how to obtain a copy. It is 338 Swiss Francs.

In the UK the standard can be obtained from the BSI. Details are given below:

http://shop.bsigroup.com/en/ProductDetail/?pid=000000000030185076

It is 356 UK pounds.

You should be able to buy the standard from the standards organisations in your country. Google is a good place to start/

1.3.2 J3 and WG5 Working Documents

Working documents can be found at the J3 and WG5 sites. The last working document for the Fortran 2003 standard can be found at both the J3 and WG5 sites. WG5 have the document available at:

ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/

and is document number n1601. It can also be found at the J3 site.

http://www.j3-fortran.org/doc/year/04/04-007.pdf

1.3.3 Compiler Documentation

The compiler may come with documentation. Here are some details for a number of compilers.

1.3.3.1 g95

A manuals is available at

http://ftp.g95.org/G95Manual.pdf

Visit

http://www.g95.org/index.shtm

for up to date information.

1.3.3.2 gfortran

Manuals are available at

http://gcc.gnu.org/wiki/GFortran#manuals

The following

http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gfortran.pdf is a 236 page pdf.

1.3.3.3 Intel

Windows. The following will end up available after a complete install.

- Intel MKL
  - Release notes
  - Reference Manual
  - User Guide
- Parallel Debugger Extension
  - Release Notes
- Compiler
  - Reference Manual, Visual Studio Help files or html.
  - User Guide, Visual Studio Help files or html.

Intel also provide the following

http://software.intel.com/en-us/articles/intel-software-technical-documentation/

1.3.3.4 Nag

Windows

- Fortran Builder Help
  - Fortran Builder Tutorial—44 pages
  - Fortran Builder Operation Guide—67 pages
  - Fortran Language Guide—115 pages
  - Compiler Manual—149 pages
  - LAPACK Guide—70 pages (440MB as PDF!)
  - GTK+Library—201 pages
  - OpenGL/GLUT Library—38 pages
  - SIMDEM Library—78 pages

1.3.3.5 Oracle/Sun

Oracle make available a range of documentation. From within Oracle Solaris Studio

- Help
  - Help Contents
  - Online Docs and Support
  - ..
  - ..
  - Quick Start Guide

and you will get taken to the Oracle site by some of these entries.

You can also download a 300+ MB zip file which contains loads of Oracle documentation. You should be able to locate (after some rummaging around)

- Sun Studio 12: Fortran Programming Guide—174 pages
- Sun Studio 12: Fortran User's Guide—216 pages
- Sun Studio 12: Fortran Library Reference—144 pages
- Fortran 95 Interval Arithmetic Programming Reference—166 pages

Happy reading :-)

1.3.4 Books

Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T.: The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures. Springer, London (2008) 31 Oct 2008, ISBN-10: 1846283787, ISBN-13: 978-1846283789.

It covers the whole of the Fortran 2003 standard in a lot of depth. The content and structure of the book follows that of the standard directly. A much easier read than the standard, and a lot cheaper.

Chapter 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 Peloponnesian 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 Carroll, Through the Looking Glass
and What Alice Found There

Aims

The aims of this chapter 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.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_2, © Springer-Verlag London Limited 2012

2.1 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.

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 to use here are:

- In natural language.
- In artificial or stylised language.

Both methods have their advantages and disadvantages.

2.2 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 3 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 a 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?

2.3 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 all will approach their problem in their own way; thus they will each use the same terminology in slightly different ways.

2.3.1 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.

2.4 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.

2.5 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 10 min. 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.

2.5.1 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 not have 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.

2.5.2 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 in 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 available. A colleague and friend of the authors 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 is 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 book. 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.

2.5.3 Stepwise Refinement

Both of the above techniques can be combined with what is called stepwise refinement. The original ideas behind this approach are well expressed in a paper by Wirth, entitled "program Development by Stepwise Refinement", published in 1971. It means that you start with a global problem statement and break the problem down in stages, into smaller and smaller subproblems 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 straightaway some thinking and pencil and paper work are 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.

2.6 Module Programming

As the problems we try solving become more complex we need to look at ways of managing the construction of programs that comprise many parts. Modula 2 was one of the first languages to support this methodology and we will look at modular programming in more depth in a subsequent chapter.

2.7 Object Oriented Programming

There is a class of problems that are best solved by the treatment of the components of these problems as objects. We will look at the concepts involved in object oriented programming and object oriented languages in the next chapter.

2.8 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 usually because the problems are reasonably simple, and it is not necessary to be explicit about 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 one's 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.

2.8.1 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 us out. It is essential that the objectives are:

- Clearly defined.
- Understood and agreed to by all people concerned, when more than one person is involved.
- Realistic.

2.8.2 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.

2.8.3 Analysis

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

2.8.4 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 stepwise 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 among several alternatives. Note that there is generally not one ideal solution, but several, each with its own advantages and disadvantages.

2.8.5 Detailed Design

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

It is at this stage that it is useful to generate pseudocode. This means writing out in detail the actions we want carried out at each stage of our overall algorithm. We gradually expand each stage (stepwise refinement) until it becomes Fortran—or whatever language we want.

2.8.6 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 enough about a programming language to use it effectively to solve our problem. This is only one stage in the overall process, and mistakes at any of the stages can create serious difficulties.

2.8.7 Evaluation and Testing

Here we try to see if the program(s) we have produced will 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 in the first place—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.

2.8.8 Maintenance

It is rare that a program is run once and never used again. This means that there will be an ongoing task of maintaining the program, generally to make it work with different versions of the operating system or 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, often with 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.

2.9 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 long time to complete. For practical purposes it is unsolvable.

Other problems can be shown mathematically to be undecidable. The work of Gödel in this area has been of enormous importance, and the bibliography contains a number of references for the more inquisitive and mathematically orientated reader. The Hofstader coverage is the easiest, and least mathematical.

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.

2.10 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, motor-cycle or bicycle having a flat tire.

2.11 Bibliography

Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading (1982)

Theoretical coverage of the design and analysis of computer algorithms.

Beckman, F.S.: Mathematical Foundations of Programming, Addison-Wesley, Reading (1981)

Good clear coverage of the theoretical basis of computing.

Bulloff, J.J., Holyoke, T.C., Hahn, S.W.: Foundations of mathematics—symposium papers commemorating the 60th birthday of Kurt Gödel, Springer-Verlag, Berlin/ Heidelberg/New York (1969)

The comment by John von Neumann highlights the importance of Gödel's work, Kurt Gödel's achievement in modern logic is singular and monumental—indeed it is more than a monument, it is a landmark which will remain visible far in space and time. Whether anything comparable to it has occurred in the logic of modern times may be debated. In any case, the conceivable proxima are very, very few. The subject of logic has certainly changed its nature and possibilities with Gödel's achievement.

Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming, Academic Press, London, New York (1972)

This is the seminal book on structured programming.

Davis, M.: Computability and Unsolvability, Dover, New York (1982)

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

Davis, W.S.: Systems Analysis and Design, Addison-Wesley, Reading (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.

Gödel, K.: On Formally Undecidable Propositions of Principia Mathematica and Related Systems. Oliver and Boyd, New York (1962)

An English translation of Gödel's original paper by Meltzer, with quite a lengthy introduction by R.B. Braithwaite, then Knightbridge Professor of Moral Philosophy at Cambridge University, England, and classified under philosophy at the library at King's, rather than mathematics.

Hofstadter, D.: The Eternal Golden Braid, Harvester Press, Hassocks (1979)

A very readable coverage of paradox and contradiction in art, music and logic, looking at the work of Escher, Bach and Gödel, respectively.

Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)

Comprehensive coverage of the theoretical basis of computing.

Kernighan, B.W., Plauger, P.J., Software Tools. Addison-Wesley, Reading (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, Reading
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, Chichester (1981)
Short and readable introduction to systems analysis and design.

Wirth, N.: Program development by stepwise refinement. Commun. ACM. **14**(4), 221–227 (1971)
Clear and simple exposition of the ideas of stepwise refinement.

Chapter 3
Introduction to Programming Languages

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

Samuel R. Delany, Babel-17

Aims

The primary aim of this chapter is to provide a short history of program language development and give some idea as to the concepts that have had an impact on Fortran. It concentrates on some but not all of the major milestones of the last 40 years, in roughly 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.

3.1 Introduction

It is important to realise that programming languages are a recent invention. They have been developed over a relatively short period—55 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 55 year span. The bulk of the chapter describes 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.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_3, © Springer-Verlag London Limited 2012

3.2 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 for most computers designed to this day.

3.3 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 is a wide range of problems and an equally wide range of programming languages, with particular languages being suited to a particular class of problems, all of which often appears bewildering to the beginner.

3.4 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 more modern ones, but most people prefer driving newer models. The same is true of programming languages, where you can achieve much with the older languages, but the newer ones are easier to use.

3.5 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 digital machines use 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.

3.5.1 Fortran's Origins

The next stage was the development of higher-level languages. The first of these was Fortran and it was developed over a 3 year period from 1954 to 1957 by an IBM team

led 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 formula translation 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.
- Was easy to teach.
- Had demonstrated the benefits of subroutines and independent compilation.
- Was relatively machine independent.
- Often had very efficient implementations.

Possibly the single most important fact about Fortran was, and still is, its widespread usage in the scientific community.

3.5.2 Fortran 77

The next standard in 1977 (actually 1978, and thus out by one—a very common programming error, more of this later!) added character handling, but little else in the way of major new features, really tidying up some of the deficiencies of the 1966 standard. One important feature sometimes overlooked was backwards compatibility. This meant that the standard did not invalidate any standard conformant Fortran 66 program. This protected investment in old code.

3.5.3 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—common Business Orientated Language. There is a need in commercial programming to describe data in a much more complex fashion than for scientific programming, and 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 among:
  - 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.

3.5.4 Algol

Another important development of the 1950s 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 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 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.

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.

3.6 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 unimportant 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.

3.7 Lisp

There were also developments in very specialized areas. List processing was proving to be of great interest in the 1950s and saw the development of IPLV between 1954 and 1958. This in turn led to the development of Lisp at the end of the 1950s. Lisp has proved to be of considerable use for programming in the areas of artificial intelligence, playing chess, automatic theorem proving and general problem 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 systems than compiled languages, they do provide great opportunities for the user to explore and try out ideas whilst sitting 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. One of the major problems for the Lisp user has been the large number of Lisp flavours, and this has reduced the impact that the language has had and deserved.

3.8 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, Ms, 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 text manipulation that Snobol's greatest contribution to program language development lies.

3.9 Second-Generation Languages

3.9.1 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 1960s 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 also true of Algol 68. The report introduced its own syntactic and semantic conventions and thus forced another stage in the learning process on the prospective user. However, it has a small but very committed user population who like the very rich facilities provided by the language.

3.9.2 Simula

Another strand that makes up program language development is provided by Simula, 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. It was effectively the first language to offer the opportunity of object orientated programming, and we will come back to this very important development in programming languages later in this chapter.

3.9.3 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.

It became the language of first choice within the field of computer science during the 1970s and 1980s, and the comment by Wirth sums up the language very well: “although Pascal had no support from industry, professional societies, or government agencies, it became widely used. The important reason for this success was that many people capable of recognising its potential actively engaged themselves in its promotion. As crucial as the existence of good implementations is the availability of documentation. The conciseness of the original report made it attractive for many teachers to expand it into valuable textbooks. Innumerable books appeared

between 1977 and 1985, effectively promoting Pascal to become the most widespread language used in introductory programming courses. Good course material and implementations are the indispensable prerequisites for such an evolution."

3.9.4 APL

APL is another interesting language of the early 1960s. It was developed by Iverson early in the decade and was available by the mid to late 1960s. 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!

3.9.5 Basic

Basic stands for Beginners All Purpose Symbolic Instruction Code, and was developed by Kemeny and Kurtz at Dartmouth during the 1960s. 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 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.

3.9.6 C

There is a requirement in computing to be able to access the underlying machine directly or at least efficiently. 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 on 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.

With the widespread use of UNIX within the academic world C gained considerable ground during the 1970s and 1980s. UNIX systems also offered much to the professional software developer, and became widely used for large-scale software development and as Ritchie says: "C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

3.10 Some Other Strands in Language Development

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

3.10.1 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.

3.10.2 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.

3.10.3 Data Structuring and Procedural Programming

By the 1970s languages started to emerge that offered the ability to organise data logically—so called data structuring, and we will look at two of these in the coverage below—C and Pascal.

C provided this facility via structs and Pascal did it via records. These languages also offered two ways of processing the data—directly or via procedures. The terms concrete and abstract data type are sometimes also used in the literature.

An example may help here. Consider a date. This is typically made up of three components, a day, a month and a year. In C we can create a user defined type called a date using structs. We can then create variables of this type. This is done in Pascal in a similar way using records.

Access to the components of a date (day, month and year) can then either be direct—an example of a concrete data type, or indirect (via procedures)—an abstract data type.

Simplistically direct access (or concrete data types) offer the benefit of efficiency, and the possibility of lack of data integrity. In our date example we may set a day to the value 31 when the month is February.

Indirect access (or abstract data types) are slightly less efficient as we now have the overhead of a procedure call to access the data, but better opportunity for data integrity as we can provide hidden code within the procedures to ensure that the day, month and year combinations are valid.

Fortran did not provide this facility until the Fortran 90 standard.

3.10.4 Standardisation

The purposes of a standard are quite varied and include:

- Investment in people: by this we mean that the time spent in learning a standard language pays off in the long term, as what one learns is applicable on any hardware/software platform that has a standard conformant compiler.
- Portability: one can take the code one has written for one hardware/software platform and move it to any hardware/software platform that has a standard conformant compiler.
- Known reference point: when making comparisons one starts with reference to the standard first, and then between the additional functionality of the various implementations

These are some but not all of the reasons for the use of standards. Their importance is summed up beautifully by Ronald G. Ross in his introduction to the Cannan and Otten book on the SQL standard: “Anybody who has ever plugged in an electric cord into a wall outlet can readily appreciate the inestimable benefits of workable standards. Indeed, with respect to electrical power, the very fact that we seldom even think about such access (until something goes wrong) is a sure sign of just how fundamentally important a successful standard can be.”

3.11 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 led 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.

3.12 Modula

Modula was designed by Wirth during the 1970s 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.

3.13 Modula 2

Wirth carried on developing his ideas about programming languages and the culmination of this can be seen in Modula 2. 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 multi-programming. 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:

- The module concept, and in particular the facility to split a module into a definition part and an implementation part.
- 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.
- The concept of process as the key to multiprogramming facilities.
- So-called low-level facilities, which make it possible to breach the rigid type consistency rules and allow one to map data with Modula 2 structure onto a store without inherent structure.
- The procedure type, which allows procedures to be dynamically assigned to variables.

A sad feature of Modula 2 has been the long time taken to arrive at a standard for the language.

3.14 Other Language Developments

The following is a small selection of language developments that the authors find interesting—they may well not be included in other people's coverage.

3.14.1 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 educationists outside of the context of Logo. Turtles have been incorporated into the Smalltalk computer system developed at Xerox Palo Alto Research Centre—Xerox PARC.

3.14.2 Postscript, TeX and LaTeX

The 1980s saw a rapid spread in the use of computers for the production of printed material. The 3 languages are each used quite extensively in this area.

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 desktop publishing systems, but underlies many of these systems. It is supported by an increasing number of laser printers and typesetters.

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.

LaTex is Leslie Lamport's version of TeX, and is regarded by many as more friendly. It is basically a set of macros that hide raw TeX from the end user. The TeX/LaTeX ratio is probably 1–9 (or so I'm reliably informed by a TeXie).

3.14.3 Prolog

Prolog was originally developed at Marseille by a group led by Colmerauer in 1972/73. It has since been extended and developed by several 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.

3.14.4 SQL

SQL stands for Structured Query Language, and was originally developed by 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.

3.14.5 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 nonnumeric 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.

It is available for most systems via anonymous FTP from a number of sites on the Internet.

3.15 Object Oriented Programming

Object oriented represents a major advance in program language development. The concepts that this introduces include:

- Classes.
- Objects.
- Messages.
- Methods.

These in turn draw on the ideas found in more conventional programming languages and correspond to

- Extensible data types.
- Instances of a class.
- Dynamically bound procedure calls.
- Procedures of a class.

Inheritance is a very powerful high-level concept introduced with object oriented programming. It enables an existing data type with its range of valid operations to form the basis for a new class, with more data types added with corresponding operations, and the new type is compatible with the original.

Fortran 2003 offered support for object oriented programming. This is achieved via the module facility rather than the class facility found in other languages like C++, Java and C#.

3.15.1 Simula

As was mentioned earlier, the first language to offer functionality in this area was Simula, and thus the ideas originated in the 1960s. The book Simula Begin by Birtwistle, Dahl, Myhrhaug and Nygaard is well worth a read as it represents one of the first books to introduce the concepts of object oriented programming.

3.15.2 Smalltalk

Language plus use of a computer system.

Smalltalk has been under development by the Xerox PARC Learning Research Group since the 1970s. 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 in the Macintosh microcomputers.

Wirth has spent some time at Xerox PARC and has been influenced by their work. In his own words "the most elating sensation was that after 16 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 original Modula 2 engine, and in the development of Oberon as both a language and an operating system.

3.15.3 Oberon and Oberon 2

As Wirth says: "The programming language Oberon is the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Several features were eliminated, and a few were added in order to increase the expressive power and flexibility of the language."

Oberon and Oberon 2 are thus developments beyond Modula 2. The main new concept added to Oberon was that of type extension. This enables the construction of new data types based on existing types and allows one to take advantage of what has already been done for that existing type.

Language constructs removed included:

- Variant records.
- Opaque types.
- Enumeration types.
- Subrange types.
- Local modules.
- WITH statement.
- type transfer functions.
- Concurrency.

The short paper by Wirth provides a fuller coverage. It is available at ETH via anonymous FTP.

3.15.4 Eiffel

Eiffel was originally developed by Interactive Software Engineering Inc. (ISE) founded by Bertrand Meyer. Meyer's book Object-Oriented Software Construction contains a detailed treatment of the concepts and theory of the object technology that led to Eiffel's design.

The language first became available in 1986, and the first edition of Meyer's book was published in 1988. The following is a quote from the Wikipedia entry.

> The design goal behind the Eiffel language, libraries, and programming methods is to enable programmers to create reliable, reusable software modules. Eiffel supports multiple inheritance, genericity, polymorphism, encapsulation, type-safe conversions, and parameter covariance. Eiffel's most important contribution to software engineering is design by contract (DbC), in which assertions, preconditions, postconditions, and class invariants are employed to help ensure program correctness without sacrificing efficiency.

3.15.5 C++

Stroustrup did his Ph.D thesis at the Computing Laboratory, Cambridge University, England, and worked with Simula. He had previously worked with Simula at the University of Aarhus in Denmark. His comments are illuminating: "but was pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased. The class and co-routine mechanisms of Simula and the comprehensive type checking mechanisms ensured that problems and errors did not (as I—and I guess most people—would have expected) grow linearly with the size of the program. Instead, the total program acted like a collection of very small (and therefore easy to write, comprehend and debug) programs rather than a single large program."

He designed C++ to provide Simula's functionality within the framework of C's efficiency, and he succeeded in this goal as C++ is a widely used object oriented programming language. The major disadvantage now concerns the largely incompatible class libraries that exist. It is hoped that the various standards bodies address this problem in the immediate future.

3.15.6 Java

Bill Joy (of Sun fame) had by the late 1980s decided that C++ was too complicated and that an object oriented environment based upon C++ would be of use. At around about the same time James Gosling (mister emacs) was starting to get frustrated with the implementation of an SGML editor in C++. Oak was the outcome of Gosling's frustration.

Sun over the next few years ended up developing Oak for a variety of projects. It wasn't until Sun developed their own web browser, Hotjava, that Java as a language hit the streets. And as the saying goes *the rest is history.*

Java is a relatively simple object oriented language. Whilst it has its origins in C++ it has dispensed with most of the dangerous features. It is OO throughout. Everything is a class.

It is interpreted and the intermediate byte code will run on any machine that has a Java virtual machine for it. This is portability at the object code level, unlike portability at the source code level—which is what we expect with most conventional languages. Some of the safe features of the language include:

- Built in garbage collection.
- No pointers—everything is passed by reference.

It is multithreaded, which makes it a delight for many applications. It has an extensive windows toolkit, the so called AWT that was in the original release of the language and Swing that came in later.

It is under continual development and at the time of writing was in its seventh major release.

3.15.7 C#

C# is a new language from Microsoft and is a key part of their .net framework. It is a modern, well-engineered language in the same family of programming languages in terms of syntax as C, C++ and Java. If you have a knowledge of one of these languages it will look very familiar.

One of the design goals was to produce a component oriented language, and to build on the work that Microsoft had done with OLE, ActiveX and COM:

- ActiveX is a set of technologies that enables software components to interact with one another in a networked environment, regardless of the language in which they were created. ActiveX was built on the Component Object Model (COM).

- COM is the object model on which ActiveX Controls and OLE are built. COM allows an object to expose its functionality to other components and to host applications. It defines both how the object exposes itself and how this exposure works across processes and networks. COM also defines the object's life cycle.
- OLE is a mechanism that allows users to create and edit documents containing items or objects created by multiple applications. OLE was originally an acronym for Object Linking and Embedding. However, it is now referred to simply as OLE. Parts of OLE not related to linking and embedding are now part of Active technology.

Other design goals included creating a language:

- where everything is an object—C# also has a mechanism for going between objects and fundamental types (integers, reals, etc.).
- Which would enable the construction of robust and reliable software—it has garbage collection, exception handling and type safety.
- Which would use a C/C++/Java syntax which is already widely known and thus help programmers converting from one of these languages to C#.

It has been updated three times since its original release. Some of the more important features added in C# 2 were Generics, Iterators, Partial Classes, Nullable Types and Static Classes. The major feature that C# 3 added for most people was LINQ, a mechanism for data querying. C# 4 was released in 2010 and added a number of additional features.

3.16 Back to Fortran!

We finish off with a coverage of the developments since the Fortran 77 standard. Practically all of the Fortran compilers available today support the Fortran 90 and 95 standards. Many also support several features of the 2003 standard, and some also implement one or more features from the Fortran 2008 standard. See the following document

http://www.fortranplus.co.uk/resources/fortran_2003_2008_compiler_support.pdf

for up to date information on what each compiler offers in terms of standard support.

3.16.1 Fortran 90

Almost as soon as the Fortran 77 standard was complete and published, work began on the next version. The language drew on many of the ideas covered in this chapter

and these help to make Fortran 90 a very promising language. Some of the new features included:

- New source form, with blanks being significant and names being up to 31 characters.
- Implicit none
- Better control structures.
- Control of the precision of numerical computation.
- Array processing.
- Pointers.
- User defined data types and operators.
- Procedures.
- Modules.
- Recursion.
- Dynamic storage allocation.

This was the major update that the Fortran community had been waiting a long time for. Backwards compatibility was again a key aim. This standard did not invalidate any standard conformant Fortran 77 program.

3.16.2 Fortran 95

Fortran was next standardised in 1996—yet again out by one! Firstly we have a clear up of some of the areas in the standard that had emerged as requiring clarification. Secondly Fortran 95 added the following major concepts:

- The forall construct.
- pure and elemental procedures.
- implicit initialisation of derived-type objects.
- Initial association status for pointers.

The first two help considerably in parallelization of code. Minor features include amongst others:

- Automatic deallocation of allocatable arrays.
- intrinsic SIGN function distinguishes between −0 and +0.
- intrinsic function NULL returns disconnected pointer.
- intrinsic function CPU_TIME returns the processor time.
- References to some pure functions are allowed in specification statements.
- Nested where constructs.
- Masked elsewhere construct.
- Small changes to the CEILING, FLOOR, MAXLOC and MINLOC intrinsic functions.

Some of these were added to keep Fortran in line with High Performance Fortran (HPF). More details are given later.

Part 2 of the standard (ISO/IEC 1539–2:1994) adds the functional specification for varying length character data type, and this extends the usefulness of Fortran for character applications very considerably.

3.16.3 ISO Technical Reports TR15580 and TR15581

There are two additional reports that have been published on Fortran. TR 15580 specifies three modules that provide access to IEEE floating point arithmetic and TR15581 allows the use of the allocatable attribute on dummy arguments, function results and structure components.

3.16.4 Fortran 2003

The language is known as Fortran 2003 even though the language did not make it through the standardisation process until 2004. It was a major revision.

- Derived-type enhancements: parameterised derived types (allows the kind, length, or shape of a derived type's components to be chosen when the derived type is used), mixed component accessibility (allows different components to have different accessibility), public entities of private type, improved structure constructors, and finalisers.
- Object oriented programming support: enhanced data abstraction (allows one type to extend the definition of another type), polymorphism (allows the type of a variable to vary at run time), dynamic type allocation, select type construct (allows a choice of execution flow depending upon the type a polymorphic object currently has), and type-bound procedures.
- The associate construct (allows a complex expression or object to be denoted by a simple symbol).
- Data manipulation enhancements: allocatable components, deferred-type parameters, volatile attribute, explicit type specification in array constructors, intent specification of pointer arguments, specified lower bounds of pointer assignment and pointer rank remapping, extended initialisation expressions, MAX and MIN intrinsics for character type, and enhanced complex constants.
- Input/output enhancements: asynchronous transfer operations (allow a program to continue to process data while an input/output transfer occurs), stream access (allows access to a file without reference to any record structure), user specified transfer operations for derived types, user specified control of rounding during format conversions, the flush statement, named constants for preconnected units, regularisation of input/output keywords, and access to input/output error messages.
- Procedure pointers.
- Scoping enhancements: the ability to rename defined operators (supports greater data abstraction) and control of host association into interface bodies.
- Support for IEC 60559 (IEEE 754) exceptions and arithmetic (to the extent a processor's arithmetic supports the IEC standard).

- Interoperability with the C programming language (allows portable access to many libraries and the low-level facilities provided by C and allows the portable use of Fortran libraries by programs written in C).
- Support for international usage: (ISO 10646) and choice of decimal or comma in numeric formatted input/output.
- Enhanced integration with the host operating system: access to command line arguments and environment variables and access to the processor's error messages (improves the ability to handle exceptional conditions).

The earlier web address has details of Fortran compiler conformance to this standard.

3.16.5 DTR 19767 Enhanced Module Facilities

The module system in Fortran has a number of shortcomings and this DTR addresses some of the issues.

One of the major issues was the so-called recompilation cascade. Changes to one part of a module forced recompilation of all code that used the module. Modula 2 addressed this issue by distinguishing between the definition or interface and implementation. This can now be achieved in Fortran via submodules.

3.16.6 Fortran 2008

The most recent standard, ISO/IEC 1539–1:2010, commonly known as Fortran 2008, was approved in September 2010. The new features include:

- Submodules—Additional structuring facilities for modules; supersedes ISO/IEC TR 19767:2005
- Coarray Fortran—a parallel execution model
- The DO CONCURRENT construct—for loop iterations with no interdependencies
- The CONTIGUOUS attribute—to specify storage layout restrictions
- The BLOCK construct—can contain declarations of objects with construct scope
- Recursive allocatable components—as an alternative to recursive pointers in derived types.

A more thorough coverage can be found in John Reid's paper.

`ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1828.pdf`

3.16.7 The Future

The two main work items for WG5 and J3 are shown below. The information is taken from the agenda from the Garching meeting—June 27—July 1, 2011 Leibniz Supercomputing Centre (LRZ), Boltzmannstr. 1 85748 Garching/Munich, Germany.

- Review the PDTR Ballot comments on the draft TR on Further Interoperability with C, decide on changes, and construct a response document.
- Consider the technical content of the proposed TR on Further Coarray Features.

There is also an effective permanent work item:

- Consider the Fortran defect reports (interpretations) in J3-006.

3.17 Internet Resources

The Internet provides access to a wealth of information regarding the Fortran family of languages.

3.17.1 Standards Information

The official home of the standard is

- http://www.nag.co.uk/sc22wg5/

We recommend visiting the site to keep up to date with Fortran developments. Their official ftp server can be found at

- ftp://ftp.nag.co.uk/sc22wg5/

Copies of all working documents can be found there.

3.17.2 Fortran Discussion Lists

The first to look at is the Fortran 90 list. Details can be found at

- http://www.jiscmail.ac.uk/lists/COMP-FORTRAN-90.html

if you subscribe you will have access to people involved in Fortran standardisation, language implementors for most of the hardware and software platforms, people using Fortran in many very specialised areas, people teaching Fortran, etc.

There is also a comp.lang.fortran list available via USENET news. This provides access to people worldwide with enormous combined expertise in all aspects of Fortran. Invariably someone will have encountered your problem or one very much like it and have one or more solutions.

There are many people on the Internet who will make the time to provide you with very valuable advice. As a point of network etiquette please do not waste bandwidth with questions that are answered in the FAQ. Please also spend some time developing an understanding of your problem and making some attempt to see if the

answer lies in the documentation or manuals. In computing services and technical support many user problems are labelled RTFM—read the fabulous manual.

3.17.3 Other Sources

The following URLs are very useful:

- Our Fortran web site.
  - http://www.fortranplus.co.uk
- The Fortran Wiki.
  - http://fortranwiki.org/
- The Fortran Market, maintained by Walt Brainerd.
  - http://www.fortran.com/fortran/market.html
- Fortran FAQ, maintained by Keith Bierman, Sun.
  - http://www.fortran.com/fortran/FAQ/cont.html

3.18 Summary

It is hoped that you now have some idea about the wide variety of uses that programming languages are put to.

3.19 Bibliography

Fortran 2008 Standard, ISO/IEC 1539–1:2010, price CHF 338. Publication date: 2010-10-06.

http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50459

Fortran 2003 Standard, ISO/IEC DIS 1539–1:2004(E)

DTR 19767: Enhanced module Facilities: ISO/IEC TR 19767:2004(E)

Fortran 77 Standard

ftp://ftp.nag.co.uk/sc22wg5/ARCHIVE/Fortran77.html

Fortran 66 Standard

ftp://ftp.nag.co.uk/sc22wg5/ARCHIVE/Fortran66.pdf

The ISO home page is

- http://www.iso.org/iso/home.htm

The J3 home page is:

- http://j3-fortran.org

The WG5 home page is:

- http://www.nag.co.uk/sc22wg5/

Both have copies of working documents.

Adobe Systems Incorporated, Postscript Language: Tutorial and Cookbook, Addison-Wesley, 1985.

Adobe Systems Incorporated, Postscript Language: Reference Manual, Addison-Wesley, 1985.

Adobe System Incorporated, Postscript Language: program Design, Addison-Wesley, 1985.

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

ACM SIG PLAN, History of programming Languages Conference—HOPL-II, ACM Press, 1993.

One of the best sources of information on C++, CLU, Concurrent Pascal, Formac, Forth, Icon, Lisp, Pascal, Prolog, Smalltalk and Simulation Languages by the people involved in the original design and or implementation. Very highly recommended. This is the second in the HOPL series, and the first was edited by Wexelblat. Details are given later.

Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin J.T., Smith, B.T.: The Fortran 2003 Handbook. Springer, London (2009)

Their most recent version, and a complete coverage of the 2003 standard. As with the Metcalf, Reid and Cohen book some of the authors were on the J3 committee. Very thorough.

Annals of the History of Computing, Special Issue: Fortran's 25 Anniversary, ACM, Article 6,1, 1984.

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

Barnes, J.: Programming in Ada 95. Addison-Wesley, Reading (1996)

One of the best Ada books. He was a member of the original design team

Bergin, T.J., Gibson, R.G.: History of Programming Languages. Addison-Wesley, New York (1996)

This is a formal book publication of the Conference Proceedings of HOPL II. The earlier work is based on preprints of the papers.

Birtwistle, G.M., Dahl, O. J., Myhrhaug, B., Nygaard, K.: Simula Begin. Chartwell-Bratt Ltd, Lund (1979)

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 Trans. Softw. Eng. **1**(2), 199–207 (June 1975)

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

Cannan, S., Otten, G.: SQL—The Standard Handbook. McGraw-Hill, McGraw-Hill (1993)

Very thorough coverage of the SQL standard, ISO 9075:1992(E).

Chivers, I.D., Clark, M.W.: History and future of Fortran. Data Process. **27**(1), (January/February 1985)

Short article on an early draft of the standard, around version 90.

Chivers Ian, Essential C# Fast, Springer, ISBN 1-85233-562-9

A quick introduction to the C# programming language.

Chivers, I.D.: A Practical Introduction to Standard Pascal. Ellis Horwood, Chichester (1986)

A short introduction to Pascal.

Date, C.J.: A Guide to the SQL Standard. Addison-Wesley, Reading (1997)

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

Deitel, H.M., Deitel, P.J.: Java: How to Program. Prentice-Hall, Upper Saddle River (1999)

A very good introduction to Java.

Deitel, H.M., Deitel, P.J., Nieto, T.R.: Simply Visual Basic .Net. Prentice-Hall, Upper Saddle River (2003)

Good practical introduction to VB .NET.

Eckstein, R., Loy, M., Wood, D.: Java Swing. O'Reilly, Sebastopol (1998)

Comprehensive coverage of the visual interface features available in Java.

Flanagan, D.: Java in a Nutshell. O'Reilly, Sebastopol (1996)

Just what you would expect from this series. Very useful reference text.

Geissman, L.B., Separate Compilation in Modula 2 and the Structure of the Modula 2 Compiler on the Personal Computer Lilith, Dissertation 7286, ETH Zurich.

Goldberg, A., Robson, D.: Smalltalk-80, The Language and Its Implementation. Addison Wesley, Reading (1983)

Harbison, S.P., Steele, G.L.: A C Reference Manual. Prentice-Hall, Englewood Cliffs (2002)

Very good coverage of the various flavours of C, including K&R C, Standard C 1989, Standard C 1995, Standard C 1999 and Standard C++

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

Fascinating background reading concerning Modula 2 and the Lilith architecture.

Goldberg, A., Robson, D.: Smalltalk 80: The Language and Its Implementation. Addison-Wesley, Reading (1983)

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, G., Hartmanis, J. (eds.), The programming Language Ada—Reference Manual. Springer Verlag, New York (1981)

The definition of the language.

Griswold, R.E., Poage, J.F., Polonsky, I.P.: The Snobol4 Programming Language. Prentice-Hall, Englewood Cliffs (1971)

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, Englewood Cliffs (1983)

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, New York (1975)

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, New York (1971)

The original book on Basic by its designers.

Kernighan, B.W., Ritchie D.M.: The C Programming Language. Prentice-Hall, Englewood Cliffs (1978)

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, Reading (1986)

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, London (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, Englewood Cliffs (1987)

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, Reading (1986)

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., Reid, J., Cohen, M.: Modern Fortran Explained. Oxford University Press, Oxford (2011)

A clear compact coverage of the main features of Fortran. John Reid is Convener of the WG5 committee and Malcolm Cohen was the editor of Fortran 2008.

Mossenbeck, H.: Object-Orientated Programming in Oberon-2. Springer-Verlag, New York (1995)

One of the best introductions to object oriented programming. Uses Oberon-2 as the implementation language. Highly recommended.

Papert, S.: Mindstorms—Children, Computers and Powerful Ideas. Harvester Press, Brighton (1980)

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

Sammet, J.: Programming Languages: History and Fundamentals. Prentice-Hall, Englewood Cliffs (1969)

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

Sethi, R.: Programming Languages: Concepts and Constructs. Addison-Wesley, Reading (1989)

The annotated bibliographic notes at the end of each chapter and the extensive bibliography make it a useful book.

Reiser, M., Wirth, N.: Programming in Oberon—Steps Beyond Pascal and Modula. Addison-Wesley, Reading (1992)

Good introduction to Oberon. Revealing history of the developments behind Oberon.

Reiser, M.: The Oberon System: User Guide and Programmer's Manual. Addison-Wesley, Reading (1991)

How to use the Oberon system, rather than the language.

Stroustroup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley, Reading (1997)

The C++ book. Written by the designer of the language. Massive improvement over the earlier editions.

Young, S.J.: An Introduction to Ada, 2nd edn. Ellis Horwood, Chichester (1984)

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

Wexelblat, R.L.: History of Programming Languages, HOPL I, ACM Monograph Series. Academic Press, New York (1978)

Very thorough coverage of the development of programming languages up to June 1978. Sessions on Fortran, Algol, Lisp, Cobol, APT, Jovial, GPSS, Simula, JOSS, Basic, PL/I, Snobol and APL, with speakers involved in the original languages. Very highly recommended.

Wiener, R.: Software Development Using Eiffel. Prentice Hall, Englewood Cliffs (1995)

Wirth, N.: An assessment of the programming language Pascal. IEEE Trans. Softw. Eng. **SE-1**(2), 192–198 (June 1975)

Wirth, N.: History and Goals of Modula 2. In: Byte, vol 9, 145–152. McGraw-Hill, Inc., Peterborough (August 1984)

Straight from the horse's mouth!

Wirth, N.: On the design of programming languages. In: Proceedings of the IFIP Congress, vol. 74, pp. 386–393. North-Holland, Amsterdam (1974)
Wirth, N.: The programming language Pascal. Acta Inform. **1**, 35–63 (1971)
Wirth, N.: Modula: a language for modular multiprogramming. Softw. Pract. Exp. **7**(1), 3–35 (1977)
Wirth, N.: Programming in Modula 2. Springer-Verlag, Berlin (1983)
The original definition of the language. Essential reading for anyone considering programming in Modula 2 on a long term basis.
Wirth, N.: Type extensions. ACM Trans. Program. Languages Syst. **10**(2), 204–214 (April 1988)
Wirth, N.: From Modula 2 to Oberon. Softw. Pract. Exp. **18**(7), 661–670 (July 1988)
Wirth, N., Gutknecht, J.: Project Oberon: The Design of an Operating System and Compiler. Addison-Wesley, Reading (1992)
Fascinating background to the development of Oberon. Highly recommended for anyone involved in large scale program development, not only in the areas of programming languages and operating systems, but more generally.

Chapter 4
Introduction to Programming

"Though this be madness, yet there is method in't" Shakespeare
'Plenty of practice' he went on repeating, all the time that Alice was getting him on his feet again. 'plenty of practice.'
The White Knight, Through the Looking Glass and What Alice Found There,

Lewis Carroll

Aims

The aims of the chapter are:

- To introduce the idea that there is a wide class of problems that can be solved with a computer and, further, that there is a relationship between the kind of problem to be solved and the choice of programming language that is used.
- To give some of the reasons for the choice of Fortran.
- To introduce the fundamental components or kinds of statements to be found in a general purpose programming language.
- To introduce the three concepts of name, type and value.
- To illustrate the above with sample programs based on three of the five intrinsic data types:
  - character, integer and real
- To introduce some of the formal syntactical rules of Fortran.

4.1 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

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_4, © Springer-Verlag London Limited 2012

languages. The reason for this is that there is a wide range of problems that are solved using a computer, e.g., the telephone company generating itemised bills 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 strongly influence your choice of programming language. Fortran stands for FORmula TRANslation, which gives a hint of the expected range of problems for which it is suitable.

4.2 Language Strengths and Weaknesses

Some of the reasons for choosing Fortran are:

- It is a modern and expressive language;
- The language is suitable for a wide class of both numeric and nonnumeric problems;
- The language is widely available on a range of hardware and operating system platforms;
- A lot of software already exists that has been written in Fortran. Some 15% of code worldwide is estimated to be in Fortran.

There are a few warts, however. Given that there has to be backwards compatibility with earlier versions some of the syntax is clumsy to say the least. However, a considerable range of problems can now be addressed quite cleanly, if one sticks to a subset of the language and adopts a consistent style.

4.3 Elements of a Programming Language

As with ordinary (so-called natural) languages, e.g., English, French, Gaelic, German, etc., programming languages have rules of syntax, grammar and spelling. The application of these rules in a programming language is 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 one first starts 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. We will look at the various kinds of statements briefly below.

4.3.1 Data Description Statements

These are necessary to describe the kinds of data that are to be processed. In the wages program, for example, there is obviously a difference between people's 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).

4.3.2 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 among 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.

4.3.3 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.

4.3.4 Input and Output (I/O) Statements

For flexibility, programs are generally written so that the data that they work on exist 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 it is 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 first name and print it out:

```
program ch0401
!
! This program reads in and prints out a name
!
implicit none
character*20 :: first_name
!
  print *,' type in your first name.'
  print *,' up to 20 characters'
  read *,first_name
  print *,first_name
!
end program ch0401
```

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, read, print.
- 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 !. 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 straightaway.
- The `implicit none` statement means that there has to be explicit typing of each and every data item used in the program. It is good programming practice to include this statement in every program that you write, as it will trap many errors, some often very subtle in their effect. Using an analogy with a play, where there is always a list of the persona involved before the main text of the play we can say that this statement serves the same purpose.
- `The character*20` 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 these data are of a certain type, and that therefore certain kinds of operations 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 first_name is to be of type `character` and only character operations are permitted. The concept of a variable is covered in the next section. Character variables of this type can hold up to 20 characters.

- 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 first_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 first_name and, in this case, what you typed in.
- The `end program` 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 period (.) ends a sentence. Note the use of the name given in the `program` statement at the start of the program.
- Note also the use of the asterisk in three different contexts.
- Indentation has been used to make the structure of the program easier to determine. Programs have to be read by human beings and we will look at this in more depth later.
- Lastly, when you do run this program, character input will terminate with the first blank character.

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.

4.4 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:

$$\text{circumference} = 2\pi r$$

This is an equation for the calculation of the circumference of a circle. The following represents a translation of this into Fortran:

```
circumference = 2 * pi * radius
```

There are a number of things to note about this equation:

- Each of the variables on the right-hand side of the equals sign (pi and radius) will have a value, which will allow the evaluation of the expression.
- When the expression is fully evaluated the value is assigned to the variable on the left-hand side of the equals sign.
- In mathematics the multiplication is implied in Fortran we have to use the * operator to indicate that we want to multiply 2 by pi by the radius.
- We do not have access to mathematical symbols like π in Fortran but have to use variable names based on letters from the Roman alphabet.

Table 4.1 Variable, type and value

Variable_name	data_type	value_stored
temperature	`real`	28.55
number_of_people	`integer`	100
first_name	`character`	Jane

Note the use of underscores to make the variable names easier to read.

The whole line is an example of an arithmetic assignment statement in Fortran.

The following arithmetic assignment statement illustrates clearly the concepts of name and value, and the difference in the equals sign in mathematics and computing:

$$\mathrm{I} = \mathrm{I} + 1$$

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,

$$\mathrm{I} = \mathrm{I} + 1$$

does not make any sense.

Variables can be of different types. Table 4.1 shows some of those available in Fortran.

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 you have written several programs.

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

```
program ch0402
!
! This program reads in three numbers and sums
! and averages them
!
implicit none
real :: n1,n2,n3,average = 0.0, total = 0.0
integer :: n = 3
  print *,' type in three numbers.'
  print *,' Separated by spaces or commas'
  read *,n1,n2,n3
  total = n1 + n2 + n3
  average = total/n
  print *,'Total of numbers is ',total
  print *,'Average of the numbers is ',average
end program ch0402
```

4.5 Notes

The program has been given a name that means something.

There are comments at the start of the program describing what it does.

The `implicit none` statement ensures that all data items introduced have to occur in a type declaration.

The next two statements are type declarations. They define the variables to be of real or integer type. Remember integers are whole numbers, whereas real numbers are those which have a decimal point. For example, 2 is an integer and 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. Note that the variables have been given sensible names to aid in making the program easier to understand.

The variables average, total and n are also given initial values within the type declaration. Variables are initially undefined in Fortran, so the variables n1, n2, n3 fall into this category, as they have not been given values at the time that they are declared.

The first `print` statement makes a text message (in this case what is between the apostrophes) appear at the terminal. As was noted 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 (n1, n2, and n3) 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 average. We could have actually used the value 3 here instead, i.e., written average = total/3 and have exactly the same effect. This would also have avoided the type declaration for n. 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.

Indentation has been used to make the structure of the program easier to determine.

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 to forget what each number means.

Finally we have the end of the program and again we have the use of the name in the `program` statement.

4.6 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 several chapters there will be a brief summary of these rules or regulations when necessary. Here are a few:

- Source is free format.
- Lowercase letters are permitted, but not required to be recognised.
- Multiple statements may appear on one line and are separated by the semicolon character.
- 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., `implicit, integer, real` or `character`.
  - Processing and I/O statements.
  - `End program` statement.
- Comments may appear anywhere in the program, after `program` and before `end`; they are introduced with a ! character, and can be in line.
- Names may be up to 63 characters in length and include the underscore character.
- Lines may be up to 132 characters.
- Up to 39 continuation lines are allowed (using the ampersand (&) as the continuation character).
- The syntax of the `read` and `print` statement introduced in these examples is
  - `read` format, input-item-list.
  - `print` format, output-item-list.
    - where format is * in the examples and called list directed format ting.
    - and input-item-list is a list of variable names separated by commas.
    - and output-item-list is a list of variable names and/or a sequence of characters enclosed in either ' or ", again separated by commas.
- if the `implicit none` statement is not used, variables that are not explicitly declared will default to `real` if the first letter of the variable name is A–H or O–Z, and to `integer` if the first letter of the variable name is I–N.

4.7 Fortran Character Set

The following summarises the Fortran character set:

Alphanumeric characters

A–Z: Uppercase letters
a–z: Lowercase letters
0–9: Digits
_: Underscore

Special characters

Graphic	Name of character
	Blank
=	Equals
+	Plus
–	Minus
*	Asterisk
/	Slash or oblique
\	Backslash
(	Left parenthesis
)	Right parenthesis
[	Left square bracket
]	Right square bracket
{	Left curly bracket
}	Right curly bracket
,	Comma
.	Period or decimal point
:	Colon
;	Semicolon
!	Exclamation mark
"	Quotation mark
%	Percent
&	Ampersand
~	Tilde
@	Commercial at
<	Less than
>	Greater than
?	Question mark
'	Apostrophe
`	Grave accent
^	Circumflex accent
|	Vertical bar or line
$	Currency symbol
#	Number sign

The default character type shall support a character set that includes the Fortran character set. By supplying non-default character types, the processor may support additional character sets. The characters available in the ASCII and ISO 10646 character sets are specified by ISO/IEC 64 6:1991 (International Reference Version) and ISO/IEC 10646–1:2000 UCS-4, respectively; the characters available in other non default character types are not specified by the standard, except that one character in each non-default character type shall be designated as a blank character to be used as a padding character.

If you live and work outside of the USA and UK you may well have problems with your keyboard when programming. There is a very good entry in Wikipedia on keyboards, that is well worth a look at for the curious.

4.8 Good Programming Guidelines

The following are guidelines, and do not form part of the Fortran language definition:

- Use comments to clarify the purpose of both sections of the program and the whole program.
- Choose meaningful names in your programs.
- Use indentation to highlight the structure of the program. Remember that the program has to be read and understood by both humans and a computer.
- Use implicit none in all programs you write to minimise errors.
- Do not rely on the rules for explicit typing, as this is a major source of errors in programming.

4.9 Compilers

A number of hardware platforms, operating systems and compilers have been used when writing this book and earlier books. The following have been used with this book:

- NAG Fortran Builder 5.1, 5.2, 5.3 for Windows
- NAG Fortran Compiler 5.1, 5.2, 5.3 for Linux.
- Intel Fortran 11.x, 12.x for Windows.
- Intel Fortran 12.x for Linux.
- gnu gfortran 4.x for Windows.
- gnu gfortran 4.x for Linux.
- Cray Fortran : Version 7.3.1—Cray Hector service
- g95 for Linux.
- pgi 10.x—Cray Hector service
- IBM XL Fortran for AIX, V13.1 (5724-X15), Version: 13.01.0000.0002
- Oracle Solaris Studio 12.0, 12.1, 12.2 for Linux

The following have been used with earlier books:

- DEC VAX under VMS and later OPEN VMS with the NAG Fortran 90 compiler.
- DEC Alpha under OPEN VMS using the DEC Fortran 90 compiler.
- Sun Ultra Sparc under Solaris:
  - NAGACE F90 compiler.
  - NAG Ware F95 compiler.

  - Sun (Release 1 .x) F90 compiler.
  - Sun (Release 2.x) F90 compiler.

- PCs under DOS and Windows:

  - DEC/Compaq Fortran 90 and Fortran 95 compilers.
  - Intel Compiler (7.x, 8.x).
  - Lahey Fujitsu Fortran 95 (5.7).
  - NAG Fortran 95 Compiler.
  - NAG Salford Fortran 90 Compiler.
  - Salford Fortran 95 Compiler.

- PCs under Linux:

  - Intel Compiler.
  - Lahey Fujitsu Fortran 95 Pro (6.1).
  - NAG Fortran 95 (4.x, 5.x).

It is very illuminating to use more than one compiler whilst developing programs.

4.10 Program Development

A number of ways of developing programs have been used, including:

- Using an integrated development environment, including

  - NAG Fortran Builder under Windows.
  - Microsoft Visual Studio with the Intel compiler under Windows.
  - Oracle Sunstudio under SuSe Linux.

- Using a DOS box and simple command line prompt under Windows.
- Using ssh to log in to the Hector service.
- Using a VPN, and SSH to log in to the IBM Power 7 system at Slovak Hydrometeorological Institute Jeséniova 17
- Using a console or terminal window under SuSe Linux.
- Using X-Windows software to log into the SUN Ultra Sparc systems.
- Using terminal emulation software to log into the SUN Ultra Sparc.
- Using DEC terminals to log into the DEC VAX and DEC Alpha systems.
- Using PCs running terminal emulation software to log into the DEC VAX and DEC Alpha systems.

It is likely that you will end up doing at least one of the above and probably more. The key stages involved are:

- Creating and making changes to the Fortran program source.
- Saving the file.
- Compiling the program:

  - if there are errors you must go back to the Fortran source and make the changes indicated by the compiler error messages.

- Linking if successful to generate an executable:
  - Automatic link. This happens behind the scenes and the executable is generated for you immediately.
  - Manual link. You explicitly invoke the linker to generate the executable.
- Running the program.
- Determining whether the program actually works and gives the results expected.

These steps must be taken regardless of the hardware platform, operating system and compiler you use. Some people like working at the operating system prompt (e.g., DOS, Linux and UNIX), and others prefer working within a development environment. Both have their strengths and weaknesses.

4.11 Problems

1. Compile and run example 1 in this chapter. Experiment with the following types of input.

   Ian
   Ian Chivers
   "Jane Margaret Sleightholme"

2. Compile and run example 2 in this chapter.
   Think about the following points:

   - Is there a difference between separating the input by spaces or commas?
   - do you need the decimal point?
   - What happens when you type in too many data?
   - What happens when you type in too few data?

   If you have access to more than one compiler repeat the above and compare the results.
3. Write a program that will read in your name and address and print them out in reverse order.
   Think about the following points:

   - How many lines are there in your name and address?
   - What is the maximum number of characters in the longest line in your name and address?
   - What happens at the first blank character of each input line?
   - Which characters can be used in Fortran to enclose each line of text typed in and hence not stop at the first blank character?
   - if you use one of the two special characters to enclose text what happens if you start on one line and then press the return key before terminating the text?

   The action here will vary between Fortran implementations.

Chapter 5
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
Round numbers are always false.

Samuel Johnsons

Aims

The aims of this chapter are to introduce:

- The rules for the evaluation of arithmetic expressions to ensure that they are evaluated as you intend.
- The idea of truncation and rounding applied to reals.
- The use of the `parameter` attribute to define or set up constants.
- The concepts and ideas involved in numerical computation, including:
  - Specifying data types using kind-type parameters.
  - The concept of numeric models and positional number systems for integer and real arithmetic and their implementation on binary devices.
  - Testing the numerical representation of different kind types on a system.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_5, © Springer-Verlag London Limited 2012

Table 5.1 Fortran operators

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

5.1 An Introduction to Arithmetic in Fortran

Most problems in the academic and scientific communities 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.

Table 5.1 lists the five arithmetic operators available in Fortran.

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

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

```
taxable_income = gross_wage - personal_allowance
cost = bill + vat + service
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:

```
tax = gross_wage - personal_allowance * tax_rate
```

This is a poorly written arithmetic expression. There is a choice of doing the subtraction before or after the multiplication. Our everyday experience says that the subtraction should take place before the multiplication. However, if this expression were evaluated in Fortran the multiplication would be done before the subtraction.

5.2 Example 1: Simple Arithmetic Expressions in Fortran

A complete program to show the correct form in Fortran is as follow:

```
program ch0501
implicit none
!
! Example of a Fortran program to calculate net pay
! given an employee's gross pay
!
real           :: gross_wage, net_wage, tax
real           :: tax_rate = 0.25
integer        :: personal_allowance = 4800
character*60   :: their_name
  print *,'Input employees name'
  read *,their_name
  print *,'Input Gross wage'
  read *,gross_wage
  tax = (gross_wage - personal_allowance) * tax_rate
  net_wage = gross_wage - tax
  print *,'Employee: ',their_name
  print *,'Gross Pay: ', gross_wage
  print *,'Tax: ',tax
  print *,'Net Pay:',net_wage
end program ch0501
```

We need to look at three areas here:

- The rules for forming expressions—the syntax.
- The rules for interpreting expressions—the semantics.
- The rules for evaluating expressions—optimisation.

The syntax rules determine which expressions are valid. The semantics determine a valid interpretation, and once this has been done the compiler can replace the expression with any other one that is mathematically equivalent, generally in the interests of optimisation.

The rules for the evaluation of expressions in Fortran are as follows:

- Brackets are used to define priority in the evaluation of an expression.
- Operators have a hierarchy of priority—a precedence. The hierarchy of operators is:
  - Exponentiation: when the expression has multiple exponentiation, the evaluation is from right to left. For example,

$$\mathrm{L = I ** J ** K}$$

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

$$\mathrm{L = I ** (J ** K)}$$

    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 rules regarding any mathematically equivalent expression means that you must use brackets to ensure the evaluation you want
  For example, with

$$A = B * C / D * E$$

  for real and complex numeric types the compiler does not necessarily evaluate in a left to right manner, i.e., evaluate B times C, then divide the result by D and finally take that result and multiply by E.
- Addition and subtraction: as for multiplication and division the rules regarding any equivalent expression apply. 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 = Mass_D/2*(Mass_A*Veloc_A/Mass_D)**2 + &
      ((Mass_A * Veloc_A)**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 ones 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 is 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.

The rule regarding any equivalent expression means if A, B and C are numeric then the following are true:

$$A + B = B + A$$
$$-A + B = B - A$$
$$A + B + C = A + (B + C)$$

The last is nominally evaluated left to right, as the additions are of equal precedence:

$$A * B = B * A$$
$$A * B * C = A * (B * C)$$

and again the last is nominally evaluated left to right, as the multiplications are of equal precedence:

$$A * B - A * C = A * (B - C)$$
$$A / B / C = A / (B * C)$$

The last is true for real and complex numeric types only.

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 to do. 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 = Mass_A * Veloc_A
Q = Mass_D / 2 * ( Moment / Mass_D ) **2 + &
   ( Moment **2) / 2
```

5.3 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.

5.3.1 Example 2: Type Conversion and Assignment

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

```
program ch0502
implicit none
real :: a,b,c
integer :: i
  a = 1.5
  b = 2.0
  c = a / b
  I = a / b
  print *,a,b
  print *,c
  print *,I
end program ch0502
```

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 an 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.

5.3.2 Example 3: Integer Division and Real Assignment

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:

```
program ch0503
implicit none
integer :: i,j,k
real :: Answer
  I = 5
  J = 2
  K = 4
  Answer = i / j * k
  print *,i
  print *,j
  print *,k
  print *,Answer
end program ch0503
```

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 an 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.

5.4 Example 4: Time Taken for Light to Travel from the Sun to Earth

How long does it take for light to reach the Earth from the Sun? Light travels $9.46 \cdot 10^{12}$ km in 1 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 h, 48 min and 45.9747 s—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 ch0504
implicit none
real :: Light_Minute, Distance, Elapse
integer :: Minute, Second
real , parameter :: Light_Year=9.46*10**12
! Light_year  : Distance travelled by light
! in one year in km
! Light_minute : Distance travelled by light
! in one minute in km
! Distance : Distance from sun to earth in km
! Elapse : Time taken to travel a
! distance (Distance) in minutes
! Minute : integer number part of elapse
! Second : integer number of seconds
! equivalent to fractional part of elapse
!
  Light_minute = Light_Year/(365.25 * 24.0 * 60.0)
  Distance = 150.0 * 10 ** 6
  Elapse = Distance / Light_minute
  Minute = Elapse
  Second = (Elapse - Minute) * 60
  print *,' Light takes ' , Minute,' Minutes'
  print *,'             ' , Second,' Seconds'
  print *,' To reach the earth from the sun'
end program ch0504
```

The calculation is straightforward; first we calculate the distance travelled by light in 1 min, 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.

Can you see any problems with this example?

5.5 The `parameter` Attribute

This statement is used to provide a way of associating a meaningful name with a constant in a program. Consider a program where π 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 π 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=4.0*atan(1.0)
.
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 attribute:

```
real , parameter :: pi=4.0*atan(1.0) , C=2.997925
```

and

```
real , parameter :: Charge=1.6021917
```

The advantage of the `parameter` attribute 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 type statement with a `parameter` attribute 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 , parameter :: parsec = 3.08*10**16 , &
                    pi = 4.0*atan(1.0) , &
                    radian = 360./pi
```

Table 5.2 Word size and integer numbers

N bits		Maximum integer
64	(2**63)–1	9,223,372,036,854,774,807
32	(2**31)–1	2,147,483,647

Table 5.3 Word size and real numbers

N bits	Precision	Smallest real largest real
64	15–18	~0.5E–308
		~0.8E+308
32	6–9	~0.3E–38
		~1.7E38

Note that access to what the hardware supports is dependent on the operating system and compiler as well

5.6 Range, Precision and Size of Numbers

The range on integer numbers and the precision and the size of floating point numbers in computing are directly related to the number of bits allocated to their internal representation. Tables 5.2 and 5.3 summarise this information for the two most common bit sizes in use for integers and reals—32 bits and 64 bits.

Table 5.2 looks at integer numbers.

Table 5.3 is a corresponding table for real numbers.

Precision is not the same as accuracy. In this age of digital timekeeping, 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:

$$130 + 12 = 140 \text{ (rounding down from 142)}$$

and similarly:

$$17 / 3 = 5.7 \text{ (rounding up from 5.666666...)}$$

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 rewrite the fraction as

$$(16 / 0.14) * 16$$

Which 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 32-bit floating point numbers 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 there are a number of physical constants (Plank constant, elementary charge, Bohr magneton etc.,) that will cause arithmetic problems due to their size. This is rarely a problem with machines with hardware support for 64-bit arithmetic.

How we get around this problem and how we move our programs from one platform to another making sure that we are working with the same precision and same range of numbers are covered in detail in the next section.

5.7 Health Warning: Optional Reading, Beginners are Advised to Leave Until Later

It is very important in scientific programming to know the range and precision of data on the hardware platform on which we are working. The facilities provided in Fortran now allow programmers to specify the range and precision they wish to use and the compiler will choose an appropriate type.

If it is not possible to offer the precision and range requested the compiler returns an error code. To avoid this happening the programmer needs to query the computer first for details of its data representations before trying to run a program which specifies range and precision.

In order to do this we use the `kind` intrinsic function, (intrinsic functions are covered in depth in Chapter 12 and Appendix C), e.g.:

```
real :: x
print *,'Kind number for X=',kind(x)
```

This will print out the kind number used by your system to represent default real variables. These kind numbers are arbitrary and there is usually no meaning attached to them.

5.7.1 Example 5: Default Kinds

Consider the following program, which demonstrates the use of the `kind` function:

```
program ch0505
implicit none
integer :: i
real :: r
character*1 :: c
logical :: l
complex :: cp
   print *,' integer   ',kind(i)
   print *,' real     ',kind(r)
   print *,' char     ',kind(c)
   print *,' logical ',kind(l)
   print *,' complex ',kind(cp)
end program ch0505
```

It is worthwhile actually typing this program in and seeing what answers you get for the system you are working on. We have examples of several compilers below.

gfortran 4.3.4, cygwin, Windows.

```
 integer                    4
 real                       4
```

```
char                    1
logical                 4
complex                 4
```

Intel 12.0.1, Windows

```
integer                   4
real                    4
char                    1
logical                 4
complex                 4
```

NAG Fortran Builder 5.3, Windows

```
integer    3
real      1
char      1
logical  3
complex  1
```

Thus it is up to each compiler implementation to decide what kind numbers are associated with each type and kind variation. Thus the kind value on its own should not be used across platforms to try to achieve portability.

In fact, specifying a kind number actually is not what is intended by the Fortran standard, so two intrinsic functions

```
selected_int_kind
```

and

```
selected_real_kind
```

are available instead. They are used to specify the range of numbers for integers and the range and precision of numbers for reals, and the compiler will return the appropriate kind numbers that it has assigned to such representations. These kind numbers can be assigned to parameters called kind type parameters, which can be used with real and integer type declarations. Let's consider the two main numeric types to see how this works.

5.7.2 Selecting Different Integer Kind Types

The Fortran standard specifies that only one integer kind needs to be available, but often a machine's architecture or compiler implementation will offer more. Most compiler implementations will offer the following:

- 8-bit or one-byte integers.
- 16-bit or two-byte integers.
- 32-bit or four-byte integers.

and 64-bit or eight byte integers will be available on certain platforms and implementations. The most common reason for choosing 8-bit or 16-bit integers is to

reduce the memory requirements of your program and the most common reason for choosing 64-bit integers is to solve specialised problems in mathematics requiring large integer numbers.

To choose an integer kind other than the default, you specify the range of the numbers you require it to lie in, in terms of a power of 10; e.g.,

```
integer, parameter :: First = selected_int_kind (2)
integer (First) :: I,J
```

selects an integer kind parameter, First, with representation which includes all integers between -10^2 and 10^2 , i.e., numbers in the range −100–100. The integer kind parameter can be used in brackets after the integer type statement to specify variables of this integer kind, e.g., I and J.

If there is no integer kind representation for the range specified, the `selected_int_kind` function returns −1. Unfortunately it is not possible to then test for −1 in a type statement, i.e., you will get a compile time error message. We suggest that you run the program in Sect. 5.7.11 to find the limits of your machine's architecture before trying to specify a kind parameter that it can't support.

5.7.3 Selecting Different Real Kind Types

The Fortran standard specifies that there must be at least two representations of the real type, the default plus one other. Often there are more, depending on what the underlying hardware can support. When working with real data there are two things to specify—range and precision. The precision is the minimum number of significant digits (all floating point numbers are normalised) to which real numbers are stored, and the range is the power of 10 of the largest number to be represented. So, for example, to specify that a variable R has a kind type that supports 15 significant figures and a range $10^{\pm 307}$ we define a real kind parameter, Long, and then use this with the `real` type declaration for R as follows:

```
integer, parameter :: Long=selected_real_kind(15,307)
real (Long) :: R
```

The only problem is if the underlying hardware can't support this specification, in which case the function will return −1 if the requested precision is unavailable, −2 if the range is unavailable, and −3 if both are unavailable. As we mentioned earlier with integer kinds, it is not possible to test for negative values in a type declaration, so before trying to use different kind types, or even just the default types, you need to know what kind types your machine supports and their range and precision.

5.7.4 Specifying Kind Types for Literal Integer and Real Constants

A literal constant is a data object whose value cannot change. An integer constant 1 is of default integer kind and a real constant 10.3 is a default real constant. If in a program you have chosen a real kind type, other than the default, then to be consistent and also to make

sure that all real arithmetic is done to the precision specified, you need to declare all real constants to be of this kind type. This is done by giving the literal constant followed by an underscore and a kind number or kind type parameter, e.g.

```
constant_kind
```

For the earlier example with a kind type parameter Long, a real literal constant of this type would be given as

```
-22.36_Long
```

It is not recommended to use the actual kind number because, as we have seen, these are not portable across machines.

The convention we use throughout this book if we require a numeric kind type other than the defaults is to specify a kind type parameter, e.g.,

```
integer, parameter :: Long = selected_real_kind (15,307)
```

and then use it with real type declarations, e.g.,

```
real (Long) :: R
```

This still doesn't make programs completely portable across different hardware platforms, so you will firstly need to run a program which tests the range of data representations. Before doing this we need to know a bit more about the underlying representation of numerical data on computer systems.

5.7.5 Positional Number Systems

Most people take arithmetic completely for granted and rarely think much about the subject. It is necessary to look at it in a bit more depth if we are to understand what the computer is doing in this area.

Our way of working with numbers is essentially a positional one. When we look at the number 1,024, for example, we rarely think of it in terms of 1 * 1,000 + 0 * 100 + 2*10 + 4*1. Thus the normal decimal system we use in everyday life is a positional one, with a base of 10.

We are probably aware that we can use other number bases, and 2, 8 and 16 are fairly common alternate number bases. As the computer is a binary device it uses base 2.

We are also reasonably familiar with a mantissa exponent or floating point combination when the numbers get very large or very small, e.g., a parsec is commonly expressed as 3.08 * 10 ** 16, and here the mantissa is 3.08, and the exponent is 10 ** 16.

The above information will help in understanding the way in which integers and reals are represented on computer systems.

5.7.6 Bit Data Type and Representation Model

The model is only defined for positive integers (or cardinal numbers), where they are represented as a sequence of binary digits, and is based on the model:

$$i = \sum_{k=0}^{n-1} b_k 2^k$$

where I is the integer value, n is the number of bits, and b_k is a bit value of 0 or 1, with bit numbering starting at 0, and reading right to left. Thus the integer 43 and bit pattern 101011 is given by:

$$43 = (1*32) + (0*16) + (1*8) + (0*4) + (1*2) + (1*1)$$

Or

$$43 = (1*2^5) + (0*2^4) + (1*2^3) + (0*2^2) + (1*2^1) + (1*2^0)$$

5.7.7 Integer Data Type and Representation Model

The integer data type is based on the model

$$i = s\sum_{k=1}^{q} l_k r^{k-1}$$

where I is the integer value, s is the sign, q is the number of digits (always positive), r is the radix or base (integer greater than 1), and l_k is a positive integer (less than r).

A base of 2 is typical so 1,023 is

$$1023 = (1*2^9) + (1*2^8) + (1*2^7) + (1*2^6) + (1*2^5) + (1*2^4) + (1*2^3) + (1*2^2) + (1*2^1) + (1*2^0)$$

5.7.8 Real Data Type and Representation Model

The real data type is based on the model

$$x = sb^e \sum_{k=1}^{m} f_k\, b^{-k}$$

where x is the real number, s is the sign, b is the radix or base (greater than 1), m is the number of bits in the mantissa, e is an integer in the range e_{min} to e_{max}, and f_k is a positive number less than b.

This means that with, for example, a 32-bit real there would be 8 bits allocated to the exponent and 24 to the mantissa. One of the bits in each part would be used to represent the sign and is called the sign bit. This reduces the number of bits that can actually be used to represent the mantissa and exponent to 31 and 7, respectively. There is also the concept of normalisation, where the exponent is adjusted so that the most significant bit is in position 22—bits are typically numbered 0–22, rather than 1–23. This form of representation is not new, and is first documented

Table 5.4 Numeric query functions

function name	Simple explanation
`kind (x)`	Returns the `kind` type
`tiny (x)`	Returns the smallest number
`huge (x)`	Returns the largest number
`precision (x)`	Returns the decimal precision
`epsilon (x)`	Smallest difference between two reals

around 1750 BC, when Babylonian mathematicians used a sexagesimal (radix 60) positional notation. It is interesting that the form they used omitted the exponent!

This is the theoretical basis of the representation of these three data types in Fortran.

This information together with the following provide a good basis for writing portable code across a range of hardware.

5.7.9 IEEE 754

The first standard IEEE 754: 1985 covered binary floating point arithmetic. The later IEEE 754: 1987 standard added decimal arithmetic.

A considerable amount of hardware now offers support for the IEEE 754 standard. The standard can be purchased from

- http://standards.ieee.org

Work is under way on the next version and you can find out details of the current state of play at

- http://grouper.ieee.org/groups/754/

There are quite a lot of good links from this site.

5.7.10 Testing the Numerical Representation of Different Kind Types on a System

You are now ready to write or adapt a program to run on your system in order to test the range of integer kind types and the range and precision of real kind types.

The following program selects several integer and real kind types and by calling the intrinsic functions KIND, HUGE, PRECISION and EPSILON produces most of the information you need to know about for these kind types. Table 5.4 provides details of what these functions do.

5.7.11 Example 6: Using the Numeric Enquiry Functions

A complete program using the above is as follows:

```
program ch0506
implicit none
!
! examples of the use of the kind
! function and the numeric inquiry functions
!
! integer arithmetic
!
! 32 bits is a common word size,
! and this leads quite cleanly
! to the following
! 8 bit integers
! -128 to 127 10**2
! 16 bit integers
! -32768 to 32767 10**4
! 32 bit integers
! -2147483648 to 2147483647 10**9
!
! 64 bit integers are increasingly available.
! this leads to
! -9223372036854775808 to
!  9223372036854775807 10**19
!
! you may need to comment out some of the following
! depending on the hardware platform and compiler
! that you use.

integer                           :: i
integer ( selected_int_kind( 2))  :: i1
integer ( selected_int_kind( 4))  :: i2
integer ( selected_int_kind( 9))  :: i3
integer ( selected_int_kind(10))  :: i4

! real arithmetic
!
! 32 and 64 bit reals are normally available.
!
! 32 bit reals 8 bit exponent, 24 bit mantissa
!
! 64 bit reals 11 bit exponent 53 bit mantissa
!
real :: r
real ( selected_real_kind( 6, 37))  :: r1
real ( selected_real_kind(15,307)) :: r2
real ( selected_real_kind(15,310)) :: r3
```

```
  print *,' '
  print *,' integer values'
  print *,' kind    huge'
  print *,' '
  print *,' ',kind(i  ),' ',huge(i  )
  print *,' '
  print *,' ',kind(i1 ),' ',huge(i1 )
  print *,' ',kind(i2 ),' ',huge(i2 )
  print *,' ',kind(i3 ),' ',huge(i3 )
  print *,' ',kind(i4 ),' ',huge(i4 )

  print *,' '
  print *,' real values'
  print *,' kind    huge                    ' , &
 'precision       epsilon'
  print *,' '
  print *,' ',kind(r),' ',huge(r),&
  '             ',precision(r),' ',epsilon(r)
  print *,' '
  print *,' ',kind(r1 ),' ',huge(r1 ),&
  '             ',precision(r1),' ',epsilon(r1)
  print *,' ',kind(r2 ),' ',huge(r2 ),&
  ' ',precision(r2),' ',epsilon(r2)
  print *,' ',kind(r3 ),' ',huge(r3 ),&
  ' ',precision(r3),' ',epsilon(r3)
end program ch0506
```

The output from the Intel compiler under Windows is:

```
  integer values
  kind    huge

               4     2147483647

               1     127
               2     32767
               4     2147483647
               8      9223372036854775807

  real values
  kind     huge                precision       epsilon

               4     3.4028235E+38
6    1.1920929E-07

               4     3.4028235E+38
6    1.1920929E-07
```

```
                8     1.797693134862316E+308
15     2.220446049250313E-016

              16
1.189731495357231765085759326628007E+4932
33
   1.925929944387235853055977942584927E-0034
```

The output from the gfortran (cygwin) compiler under Windows is:

```
  integer values
  kind     huge

                4     2147483647

                1     127
                2     32767
                4     2147483647
                8     9223372036854775807

  real values
  kind     huge                 precision       epsilon

                4     3.40282347E+38
6     1.19209290E-07

                4     3.40282347E+38
6     1.19209290E-07
                8     1.7976931348623157E+308
15     2.2204460492503130E-016
               10                              +Infinity
18     1.0842021724855044340E-0019
```

The output from the same compiler under SuSe Linux, same dual boot system.

```
  Integer values
  Kind     Huge

                4      2147483647

                1      127
                2      32767
                4      2147483647
                8      9223372036854775807
```

```
  Real values
  Kind     Huge                    Precision
epsilon

                4       3.40282347E+38
6     1.19209290E-07

                4       3.40282347E+38
6     1.19209290E-07
                8       1.7976931348623157E+308
15
2.2204460492503131E-016
               10       1.189731495357231765021E+4932
18
1.08420217248550443401E-0019
```

The NAG Fortran Builder output:

```
  integer values
  kind     huge

    3    2147483647

    1    127
    2    32767
    3    2147483647
    4    9223372036854775807

  real values
  kind     huge                  precision        epsilon

    1      3.4028235E+38                6
1.1920929E-07

    1      3.4028235E+38                6
1.1920929E-07
    2     1.7976931348623157E+308  15
2.2204460492503131E-16
```

The Oracle Solaris Studio output:

```
   Integer values
   Kind Huge

    4    2147483647
```

```
     1    127
     2    32767
     4    2147483647
     8    9223372036854775807

   Real values
   Kind     Huge                Precision
epsilon

     4    3.4028234E+38              6   1.1920929E-7

     4    3.4028234E+38              6   1.1920929E-7
     8    1.7976931348623157E+308  15
2.220446049250313E-16
  16     1.189731495357231765085759326628007E+4932
33
  1.925929944387235853055977942584927 3E-34
```

Run this program on whatever system you have access to and compare the output with the above examples.

5.7.12 *Example 7: Binary Representation of Different Integer Kind Type Numbers*

For those who wish to look at the internal binary representation of integer numbers with a variety of kinds, we have included the following program

`selected_int_kind`(2) means provide at least an integer representation with numbers between–10^2 and + 10^2.
`selected_int_kind`(4) means provide at least an integer representation with numbers between–10^4 and + 10^4.
`selected_int_kind`(9) means provide at least an integer representation with numbers between-10^9 and + 10^9.

We use the `int` function to convert from one integer representation to another.

We use the logical function `btest` to determine whether the binary value at that position within the number is a zero or a one, i.e., if the bit is set.

I_in_Bits is a character string that holds a direct mapping from the internal binary form of the integer and a text string that prints as a sequence of zeros or ones:

```
program ch0507
!
! use the bit functions in Fortran to write out a
! 32 bit integer number as a sequence of
! zeros and ones
!
implicit none
integer :: j
integer :: i
integer ( selected_int_kind( 2)) :: i1
integer ( selected_int_kind( 4)) :: i2
integer ( selected_int_kind( 9)) :: i3
character (len=32) :: i_in_Bits
   print *,' type in an integer '
   read * , i
   i1=int(i,kind(2))
   i2=int(i,kind(4))
   i3=int(i,kind(9))
   i_in_Bits=' '
   do j=0,7
      if (btest(i1,j)) then
        i_in_Bits(8-J:8-J)='1'
      else
        i_in_Bits(8-J:8-J)='0'
      end if
   end do
   print *,'         1         2         3'
   print *,'12345678901234567890123456789012345678901234567890'
   print *,i1
   print *,i_in_Bits
   do j=0,15
     if (btest(i2,j)) then
       i_in_bits(16-j:16-j)='1'
     else
       i_in_bits(16-j:16-j)='0'
     end if
   end do
   print *,i2
   print *,i_in_bits
   do j=0,31
     if (btest(i3,j)) then
       i_in_bits(32-j:32-j)='1'
     else
       i_in_bits(32-j:32-j)='0'
     end if
   end do
```

```
  print *,i3
  print *,i_in_Bits
end program ch0507
```

The do loop indices follow the convention of an 8-bit quantity starting at bit 0 and ending at bit 7, 16-bit quantities starting at 0 and ending at 15, etc.

The numbers written out follow the conventional mathematical notation of having the least significant quantity at the right-hand end of the digit sequence, i.e., with 127 in decimal we have 1 * 100, 2*10 and 7 * 1, so 00100001 in binary means 1 * 32 + 1 * 1 decimal.

Try running this program on the system you are using. Does it produce the results you expect? Experiment with a variety of numbers. Try at least the following 0, + 1, –1, –128, 127, 128, –32768, 32767, 32768.

5.7.13 Example 8: Binary Representation of a Real Number

The following program is a simple variant of the previous one, but we now look at a floating point number:

```
program ch0508
!
! use the bit functions in Fortran to write out a
! 32 bit integer number equivalenced to a real
! using the transfer intrinsic as a sequence of
! zeros and ones
!
implicit none
integer                    :: i,j
character (len=32)         :: i_in_Bits=" "
real                       :: x=-1.0
  print *,'           1           2           3'
  print *,'12345678901234567890123456789012345678901234567890'
  print *,i_in_Bits
  i=transfer(x,i)
  do j=0,31
    if (btest(i,j)) then
      i_in_Bits(32-j:32-j)='1'
    else
      i_in_Bits(32-j:32-j)='0'
    end if
  end do
  print *,x
  print *,i_in_Bits
end program ch0508
```

We use the intrinsic function transfer to help out here. The `btest` intrinsic takes an integer argument, so we need to copy the bit pattern of the real number into an integer variable.

5.7.14 Summary of How to Select the Appropriate Kind Type

To write programs that will perform arithmetically in a similar fashion on a variety of hardware requires an understanding of:

- The integer data representation model and in practice the word size of the various integer kind types.
- The real data representation model and in practice the word size of the various real kind types and the number of bits in both the mantissa and exponent.

Armed with this information we can then choose a kind type that will ensure minimal problems when moving from one platform to another. End of health warning!

5.8 Variable Status

Fortran has two concepts regarding the status of a variable: defined and undefined. If a program does not provide an initial value (in a type statement) for a variable then its status is said to be undefined. Consider the following code segment taken from the earlier example that calculated the sum and average of three numbers:

```
real :: N1, N2, N3, Average=0.0, Total=0.0
integer :: N=3
```

In the above the variables Average, Total and N all have a defined status. However, N1, N2 and N3 are said to be undefined. The use of undefined values is implementation dependent and therefore not portable. Care must be taken when writing programs to ensure that your variables have a defined status wherever possible. We will look at this area again in subsequent chapters.

5.9 Summary

The following are some practical rules and guidelines:

- 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 owing 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 owing to the assignment statement when there is an integer on the left-hand side 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.
- Learn what the default `kinds` are for the numeric types you work with, what the maximum and minimum values and precision are for `real` data, and what the maximum and minimum are for `integer` data.
- You have been introduced to the use of the functions `digits, huge` and `precision`, and some of the concepts involved in their use. We will look at functions in much greater depth later on.

5.10 Problems

1. Compile and run examples 1 through 3 in this chapter.
2. Have another look at example 4. Compile and run it. It will generate an error on some systems. Can you see where the error is?
3. Write a program to calculate the period of a pendulum. This is given mathematically as

$$t = 2\pi\sqrt{\frac{length}{9.81}}$$

use the following Fortran arithmetic assignment statement:

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

The length (LENGTH) is in metres, and the time (T) in seconds. π was given a value earlier in this chapter.

Repeat the above using two other methods. Try a hand-held calculator and a spreadsheet. Do you get the same answers?

4. Base conversion.
In this chapter you have seen a brief coverage of base conversion. The following program illustrates some of the problems that can occur when going from base 10 to base 2 and back again. Which numbers will convert without loss?

```
program base_conversion
implicit none
real :: x1=1.0
real :: x2=0.1
real :: x3=0.01
real :: x4=0.001
```

```
real :: x5=0.0001
  print *,' ',x1
  print *,' ',x2
  print *,' ',x3
  print *,' ',x4
  print *,' ',x5
end program base_conversion
```

Which do you think will provide the same number as originally entered?

5. Simple subtraction. In this chapter we looked at representing floating point numbers in a finite number of bits.
   Try the following program:

```
program subtract
implicit none
real :: a=1.0002
real :: b=1.0001
real :: c
   c=a-b
   print *,a
   print *,b
   print *,c
end program subtract
```

6. Expression equivalence. We introduced some of the rules that apply in Fortran for expression evaluation. In mathematics the following is true:

$$\left(x^2 - y^2\right) = \left(x * x - y * y\right) = \left(x - y\right) * \left(x + y\right)$$

   Try the following program:

```
program expression_equivalence
!
! simple evaluation of x*x-y*y
! when x and y are similar
!
! we will evaluate in three ways.
!
implicit none
real :: x=1.002
real :: y=1.001
real :: t1,t2,t3,t4,t5
   t1=x-y
   t2=x+y
   print *,t1
   print *,t2
```

```
    t3=t1*t2
    t4=x**2-y**2
    t5=x*x-y*y
    print *,t3
    print *,t4
    print *,t5
end program expression_equivalence
```

Solve the problem with pencil and paper, calculator and Excel.
The last three examples show that you must be careful when using a computer to solve problems.

7. The following is a simple variant of ch0504. In this case we initialise light year in an assignment statement. do you think you will get the same results as from running the earlier example?

```
program ch0504p
implicit none
real :: Light_Minute, Distance, Elapse
integer :: Minute, Second
real :: Light_Year
! Light_year : Distance travelled by light
! in one year in km
! Light_minute : Distance travelled by light
! in one minute in km
! Distance : Distance from sun to earth in km
! Elapse : Time taken to travel a
! distance (Distance) in minutes
! Minute : integer number part of elapse
! Second : integer number of seconds
! equivalent to fractional part of elapse
!
  Light_Year = 9.46*10**12
  Light_minute = Light_Year/(365.25 * 24.0 * 60.0)
  Distance = 150.0 * 10 ** 6
  Elapse = Distance / Light_minute
  Minute = Elapse
  Second = (Elapse - Minute) * 60
  print *,' Light takes ' , Minute,' Minutes'
  print *,'             ' , Second,' Seconds'
  print *,' To reach the earth from sun'
end program ch0504p
```

5.11 Bibliography

Some understanding of numerical analysis is essential for successful use of Fortran when programming. As Froberg says "numerical analysis is a science—computation is an art." The following are some of the more accessible books available.

Froberg, C.E.: Introduction to Numerical Analysis. Addison-Wesley, Reading (1969)

The short chapter on numerical computation is well worth a read; it covers some of the problems of conversion between number bases and some of the errors that are introduced when we compute numerically. The Samuel Johnson quote owes its inclusion to Froberg!

IEEE, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754–2008, Institute of Electrical and Electronic Engineers Inc.

The formal definition of IEEE 754. This is available for purchase at http://www.techstreet.com/standards/ieee/754_2008?product_id=1745167 as both a pdf and printed version.

Knuth, D.: Seminumerical Algorithms. Addison-Wesley, Reading (1969)

A more thorough and mathematical coverage than Wakerly. The chapter on positional number systems provides a very comprehensive historical coverage of the subject. As Knuth points out the floating point representation for numbers is very old, and is first documented around 1750 B.C. by Babylonian mathematicians. Very interesting and worthwhile reading.

Sun, Numerical Computation Guide, SunPro (1993)

Very good coverage of the numeric formats for IEEE Standard 754 for Binary Floating-Point Arithmetic. All SunPro compiler products support the features of the IEEE 754 standard.

Wakerly, J.F.: Microcomputer Architecture and Programming. Wiley, New York (1981)

The chapter on number systems and arithmetic is surprisingly easy. There is a coverage of positional number systems, octal and hexadecimal number system conversions, addition and subtraction of nondecimal numbers, representation of negative numbers, two's complement addition and subtraction, one's complement addition and subtraction, binary multiplication, binary division, bcd or binary coded decimal representation and fixed and floating point representations. There is also coverage of a number of specific hardware platforms, including DEC PDP-11, Motorola 68000, Zilog Z8000, TI 9900, Motorola 6809 and Intel 8086. A little old but quite interesting nevertheless.

Chapter 6
Arrays 1: Some Fundamentals

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

William Shakespeare, The Sonnets

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 the chapter are to introduce the fundamental concepts of arrays and do loops, in particular:

- To introduce the idea 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 and to note that the data in an array are 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.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_6, © Springer-Verlag London Limited 2012

6.1 Tables of Data

Consider the examples below.

6.1.1 Telephone Directory

A telephone directory consists of the following kinds of entries:

Name	Address	Number
Adcroft A.	61 Connaught Road, Roath, Cardiff	223309
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 three 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.

6.1.2 Book Catalogue

A Catalogue Could Contain:

Author(s)	Title	Publisher
Carroll L.	Alice through the looking glass	Penguin
Steinbeck J.	Sweet Thursday	Penguin
Wirth N.	Algorithms plus data structures = program	Prentice-Hall

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

6.1.3 Examination Marks or Results

This could consist of:

Name	Physics	Maths	Biology	History	English	French
Fowler L.	50	47	28	89	30	46
Barron L. W	37	67	34	65	68	98
Warren J.	25	45	26	48	10	36
Mallory D.	89	56	33	45	30	65
Codd S.	68	78	38	76	98	65

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

6.1.4 Monthly Rainfall

The following data are a sample of monthly average rainfall for London in inches:

Month	Rainfall
January	3.1
February	2.0
March	2.4
April	2.1
May	2.2
June	2.2
July	1.8
August	2.2
September	2.7
October	2.9
November	3.1
December	3.1

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 are being considered have 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 the majority of books on Fortran programming, is array.

6.2 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.

6.2.1 The `dimension` Attribute

The dimension attribute 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:

```
real , dimension(1:100) :: Wages
integer , dimension(1:10000) :: Sample
```

For the variable Wages it is of type `real` and an array of dimension or size 100, i.e., the variable array Wages can hold up to 100 real items.

For the variable Sample it is of type `integer` and an array of dimension or size 10,000, i.e., the variable Sample can hold up to 10,000 integer items.

6.2.2 An Index

An index enables you to refer to or select individual elements of the array. In the telephone directory, book catalogue, exam marks table and monthly rainfall examples we used the name to index or look up the items of interest. We will give concrete Fortran code for this in the example of monthly rain fall.

6.2.3 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.

6.3 Example 1: 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	RainFall(1)	3.1
February	2	RainFall(2)	2.0
March	3	RainFall(3)	2.4
April	4	RainFall(4)	2.1
May	5	RainFall(5)	2.2
June	6	RainFall(6)	2.2
July	7	RainFall(7)	1.8
August	8	RainFall(8)	2.2
September	9	RainFall(9)	2.7
October	10	RainFall(10)	2.9
November	11	RainFall(11)	3.1
December	12	RainFall(12)	3.1

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 1/3/2000, for 1st March 2000 (anglicised style) or January 3rd (americanised style). Fortran, in common with other 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 is an example of an assignment statement showing the use of an index:

```
RainFall(1)=3.1
```

We saw earlier that we could use the `dimension` attribute to indicate that a variable was an array. In the above example Fortran statement our array is called RainFall. In this statement we are assigning the value 10.4 to the first element of the array; i.e., the rainfall for the month of January is 10.4. We use the index 1 to represent the first month. Consider the following statement:

```
SummerAverage = (RainFall(6)+RainFall(7)+&
                 RainFall(8))/3
```

This statement says take the values of the rainfall for June, July and August, add them up and then divide by 3, and assign the result to the variable SummerAverage, thus providing us with the rainfall average for the three summer months—Northern Hemisphere of course.

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 ch0601
implicit none
real :: Total=0.0, Average=0.0
real , dimension(1:12) :: RainFall
integer :: Month
  print *,' type in the rainfall values'
  print *,' one per line'
  do Month=1,12
    read *, RainFall(Month)
  enddo
  do Month=1,12
    Total = Total + RainFall(Month)
  enddo
  Average = Total / 12
  print *,' Average monthly rainfall was'
  print *, Average
end program ch0601
```

RainFall 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 Counter = Start, End, Increment
```

The block of statements that form the loop is contained between the `do` statement, which marks the beginning of the block or loop, and the `enddo` statement, which marks the end of the block or loop.

In this program, the do loops take the form:

```
do Month=1,12          start
   …                        body
enddo                  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 example above, both loops would be executed 12 times. In both cases, the first time around the loop the variable MONTH would have the value 1, the second time around the loop the variable MONTH would have the value 2, etc., and the last time around the loop MONTH would have the value 12.

A summation:

$$\sum_{i=1}^{i=12} x_i$$

is often expressed in Fortran as a loop as in this example:

```
do Month=1,12
  Total = Total + RainFall(Month)
enddo
```

6.3.1 Possible Missing Data

The rainfall data in this example has been taken from the UK Met Office site. Visit

http://www.metoffice.gov.uk/climate/uk/stationdata

to see where some of the stations are. One of us was born in Wales, the other in Yorkshire so we have chosen stations accordingly.

The following is one of the mid Wales stations:

http://www.metoffice.gov.uk/climate/uk/stationdata/cwmystwythdata.txt

Here is a sample of data from this site for 2 years.

yyyy	mm	tmax degC	tmin degC	af days	rain mm	sun hours
1959	1	4.5	-1.9	20	---	57.2
1959	2	7.3	0.9	15	---	87.2
1959	3	8.4	3.1	3	---	81.6
1959	4	10.8	3.7	1	---	107.4
1959	5	15.8	5.8	1	---	213.5
1959	6	16.9	8.2	0	---	209.4
1959	7	18.5	9.5	0	---	167.8
1959	8	19.0	10.5	0	---	164.8
1959	9	18.3	5.9	0	---	196.5
1959	10	14.8	7.9	1	---	101.1
1959	11	8.8	3.9	3	---	38.9
1959	12	7.2	2.5	3	---	19.2
1961	1	5.4	0.2	11	144.8	31.0
1961	2	8.7	2.9	2	112.5	45.2
1961	3	10.2	2.1	10	77.2	102.6
1961	4	11.9	5.0	1	130.7	83.9
1961	5	---	---	---	66.3	173.7

(continued)

(continued)

yyyy	mm	tmax degC	tmin degC	af days	rain mm	sun hours
1961	6	---	7.4	---	66.1	190.6
1961	7	16.7	8.2	0	141.1	149.2
1961	8	16.8	10.1	0	149.5	106.6
1961	9	17.4	9.3	0	134.8	79.7
1962	5		4.2	3	117.8	102.2
1962	6		6.8	1	72.8	163.9
1962	7	16.8	9.1	0	56.7	-
1962	8	15.6	9.3	0	236.2	-
1962	9	14.6	7.8	1	218.0	-
1962	10	---	---	---	69.7	-
1962	11	7.6	1.8	9	85.2	-
1962	12	5.3	-1.0	18	204.4	-

Wales is relatively wet for the UK!
The following station is Whitby:

http://www.metoffice.gov.uk/climate/uk/stationdata/whitbydata.txt

Here is a sample of the Whitby data.

YYYY	mm	tmax degC	tmin degC	af days	rain mm	sun hours
1968	1	6.9	1.7	12	24.4	
1968	2	4.3	-0.7	16	45.1	
1968	3	9.4	3.4	2	34.5	
1968	4	10.8	1.6	9	28.8	
1968	5	10.6	2.8	2	37.1	
1968	6	16.7	6.8	0	58.5	
1968	7	15.0	8.1	0	81.4	
1968	8	16.3	9.6	0	28.0	
1968	9	15.7	---	---	66.0	
1968	10	14.7	---	---	35.2	
1968	11	8.5	5.1	1	35.1	
1968	12	5.7	1.5	9	-	
1969	1	7.3	2.2	6	48.4	
1969	2	3.1	-0.8	14	46.3	
1969	3	4.5	0.4	9	-	
1969	4	8.9	2.9	4	52.6	
1969	5	11.9	6.4	0	73.7	
1969	6	16.0	8.2	0	53.0	
1969	7	19.6	11.9	0	39.0	
1969	8	17.7	12.2	0	20.6	
1969	9	16.5	10.3	0	49.2	
1969	10	15.4	9.0	0	9.0	
1969	11	7.9	2.2	4	77.2	
1969	12	5.8	1.4	9	64.1	

Bram Stoker found some of his inspiration for Dracula after staying in the town.

If you look at the data for some of these stations you will notice that data is missing for some months.

How do you think you could cope with missing data in Fortran?

The SQL standard has the concept of nulls or missing values, and missing data in a statistics package is commonly flagged by an exceptional value e.g. -999.

6.4 Example 2: People's Weights and Setting the Array Size with a Parameter

In the table below 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 and it is used to pick out or select elements of the array. The table is read as the first element of the array WEIGHT has the value 85, the second element has the value 76, etc.

Person	Associated integer representation	Array and index	Associated value
Andy	1	Weight(1)	85
Barry	2	Weight(2)	76
Cathy	3	Weight(3)	85
Dawn	4	Weight(4)	90
Elaine	5	Weight(5)	69
Frank	6	Weight(6)	83
Gordon	7	Weight(7)	64
Hannah	8	Weight(8)	57
Ian	9	Weight(9)	65
Jatinda	10	Weight(10)	76

In the first example we so-called hard coded the number 12, which is the number of months, into the program. It occurred four times. Modifying the program to work with a different number of months would obviously be tedious and potentially error prone.

In this example we parameterise the size of the array and reduce the effort involved in modifying the program to work with a different number of people:

```
program ch0602
! The program reads up to number_of_people weights
! into the array Weight
! Variables used
! Weight, holds the weight of the people
! Person, an index into the array
```

```
! Total, total weight
! Average, average weight of the people
! Parameters used
! NumberOfPeople ,10 in this case.
! The weights are written out so that
! they can be checked
!
implicit none
integer , parameter :: number_of_people = 10
real :: total = 0.0, average = 0.0
integer :: person
real , dimension(1:number_of_people) :: weight
  do person=1,number_Of_people
    print *, ' type in the weight for person ',person
    read *,weight(person)
    total = total + weight(person)
  enddo
  average = total / number_of_people
  print *,' The total of the weights is ',total
  print *,' Average Weight is ',average
  print *,' ',number_of_people,' Weights were '
  do person=1,number_of_people
    print *,weight(person)
  enddo
end program ch0602
```

6.5 Summary

The `dimension` attribute declares a variable to be an array, and must come at the start of a program unit, with other declarative statements. It has two forms and examples of both of them are given below. In the first case we explicitly specify the upper and lower :

```
real , dimension(1:number_of_people) :: Weight
```

In the second case the lower limit defaults to 1

```
real , dimension(number_of_people) :: Weight
```

The latter form will be seen in legacy code, especially Fortran 77 code suites.

The `parameter` attribute declares a variable to have a fixed value that cannot be changed during the execution of a program. In our example above note that this statement occurs before the other declarative statements that depend on it. To recap the statements covered so far, the order is summarised below.

program	First statement	
integer real character	Declarative	In any order and the dimension and parameter attributes are added here
Arithmetic assignment print * read * do enddo	Executable	In any order
end program	Last statement	

We choose individual members using an index, and these are always of integer type in Fortran.

The `do` loop is a very convenient control structure for manipulating arrays, and we use indentation to clearly identify loops.

6.6 Problems

1. Compile and run example 1 from this chapter. If you live in the UK visit the Met Office site mentioned earlier and choose a site near you, and a year of interest, making sure that the data set is complete for that year.
   If you don't live in the UK is there a site similar to the Met Office site that has data for the country you are from?
2. Compile and run program 2.
3. Using a `do` loop and an array rewrite the program which calculated the average of three numbers to ten.

4.1 Modify the program that calculates the total and average of people's weights to additionally read in their heights and calculate the total and average of their heights. Use the data given below, which have been taken from a group of first year undergraduates:

Height	Weight
1.85	85
1.80	76
1.85	85
1.70	90
1.75	69
1.67	83
1.55	64
1.63	57
1.79	65
1.78	76

4.2 Your body mass index is given by your weight (in kilos) divided by your height (in metres) squared. Calculate and print out the BMI for each person.
Grades of obesity according to Garrow as follows:

Grade 0 (desirable) 20–24.9
Grade 1 (overweight) 25–29.9
Grade 2 (obese) 30–40
Grad 3 (morbidly obese) >40
Ideal BMI range,
Men, Range 20.1–25 kg/m^2
Women, Range 18.7–23.8 kg/m^2

4.3 When working on either a UNIX system or a PC in a DOS box it is possible to use the following characters to enable you to read data from a file or write output to a file when running your program:

character	Meaning
<	read from file
>	write to file

On a typical UNIX system we could use

```
a.out < data.txt > results.txt
```

to read the data from the file called data.txt and write the output to a file called results.txt.

On a PC in a DOS box the equivalent would be

```
program.exe < data.txt > results.txt
```

This is a quick and dirty way of developing programs that do simple I/O; we don't have to keep typing in the data and we also have a record of the behaviour of the program. Rerun the program that prints out the BMI values to write the output to a file called results.txt. Examine this file in an editor.

5. Modify the program that read in your name to read in ten 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.

   Hint: Look at the formal syntax of the do statement.

6. Modify the rainfall program (which assumes that the measurement is in inches) to convert the values to centimetres. One inch equals 2.54 cm. Print out the two sets of values as a table.

   Hint: use a second array to hold the metric measurements.

7. Combine the programs that read in and calculate the average weight with the one that reads in peoples names. The program should read 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.

8. In an earlier chapter we used the following formula to calculate the period of a pendulum:

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

write a program that uses a do loop to make the length go from 1 to 10 m in 1 m increments.

Produce a table with two columns, the first of lengths and the second of periods.

Chapter 7
Arrays 2: Further Examples

Sir, In your otherwise beautiful poem (The Vision of Sin) there is a verse which reads
Every moment dies a man,
every moment one is born.
Obviously this cannot be true and I suggest that in the next edition you have it read
Every moment dies a man,
every moment 1 1/16 is born.
Even this value is slightly in error but should be sufficiently accurate for poetry.

Charles Babbage in a letter to Lord Tennyson

Aims

The aims of the chapter are to extend the concepts introduced in the previous chapter and in particular:

- To set an array size at run time – allocatable arrays.
- To introduce the idea of an array with more than one dimension and the corresponding control structure to permit easy manipulation of higher-dimensioned arrays.
- To introduce an extended form of the dimension attribute declaration, and the corresponding alternative form to the do statement, to 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.
- To formally define the block do syntax.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_7, © Springer-Verlag London Limited 2012

7.1 Varying the Array Size at Run Time

The earlier examples set the array size in the following two ways:

- Explicitly using a numeric constant
- Implicitly using a parameterised variable

In both cases we knew the size of the array at the time we compiled the program. We may not know the size of the array at compile time and Fortran provides the allocatable attribute to accommodate this kind of problem.

7.1.1 Example 1: Allocatable Arrays

Consider the following example.

```
program ch0701
!
! This program is a simple variant of ch0602.
! The array is now allocatable
! and the user is prompted for the
! number of people at run time.
!
implicit none
integer :: Number_Of_People
real ::  Total = 0.0, Average = 0.0
integer :: Person
real , dimension(:) , allocatable :: Weight
  print *,' How many people?'
  read *,Number_Of_People
  allocate (Weight(1:Number_Of_People))
  do Person=1,Number_Of_People
    print *, ' type in the weight for person ',Person
    read *,Weight(Person)
    Total = Total + Weight(Person)
  enddo
  Average = Total / Number_Of_People
  print *,' The total of the weights is ',Total
  print *,' Average Weight is ',Average
  print *,' ',Number_of_People,' Weights were '
  do Person=1,Number_Of_People
    print *,Weight(Person)
  enddo
end program ch0701
```

The first statement of interest is the type declaration with the dimension and allocatable attributes, e.g.,

```
real , dimension(:) , allocatable :: Weight
```

The second is the allocate statement where the value of the variable Number_of_people is not known until run time, e.g.,

```
allocate(Weight(1:Number_Of_People))
```

We will look more formally at these statements in Chap. 8.

7.2 Higher-Dimension Arrays

There are many instances where it is necessary to have arrays with more than one dimension. Consider the examples below.

7.2.1 Example 2: Two Dimensional Arrays and a Map

Consider the representation of the height of an area of land expressed as a two dimensional table of numbers e.g., we may have some information represented in a simple table as follows:

	Longitude		
Latitude	1	2	3
1	10.0	40.0	70.0
2	20.0	50.0	80.0
3	30.0	60.0	90.0

The values in the array are the heights above sea level. The example is obviously artificial, but it does highlight the concepts involved. For those who have forgotten their geography, lines of latitude run east–west (the equator is a line of latitude) and lines of longitude run north–south (they go through the poles and are all of the same length). In the above table therefore the latitude values are ordered by row and the longitude values are ordered by column.

A program to manipulate this data structure would involve something like the following:

```
program ch0702
! Variables used
! Height - used to hold the heights above sea level
! Long - used to represent the longitude
! Lat - used to represent the latitude
!    both restricted to integer values.
```

```
! Correct - holds the correction factor
implicit none
integer , parameter :: n = 3
integer :: Lat , Long
real , dimension(1:n,1:n) :: Height
real , parameter :: Correct = 10.0
  do Lat = 1,n
    do Long = 1,n
      print *,' type in value at ',Lat,' ',Long
      read * , Height(Lat,Long)
    enddo
  enddo
  do Lat = 1,n
    do Long = 1,n
      Height(Lat,Long) = Height(Lat,Long) + Correct
    enddo
  enddo
  print * , ' Corrected data is '
  do Lat = 1,n
    do Long = 1,n
      print * , Height(Lat,Long)
    enddo
  enddo
end program ch0702
```

Note the way in which indentation has been used to highlight the structure in this example. Note also the use of a textual prompt to highlight which data value is expected. Running the program highlights some of the problems with the simple I/O used in the example above. We will address this issue in the next 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 is an example of a valid nested do loop:

```
    do              ! Start of outer loop
       do              ! Start of inner loop

          .
          .
       enddo           ! End of inner loop
    enddo           ! End of outer loop
```

This example introduces the concept of two indices, and can be thought of as a row and column data structure.

7.2.2 Example 3: Sensible Tabular Output

The first example had the values printed in a format that wasn't very easy to work with. In this example we introduce a so-called implied do loop, which enables us to produce neat and humanly comprehensible output:

```
program ch0703
! Variables used
! Height - used to hold the heights above sea level
! Long - used to represent the longitude
! Lat - used to represent the latitude
!    both restricted to integer values.
implicit none
integer , parameter :: n = 3
integer  :: Lat , Long
real , dimension(1:n,1:n) :: Height
real , parameter :: Correct = 10.0
  do Lat = 1,n
    do Long = 1,n
     read * , Height (Lat,Long)
     Height(Lat,Long) = Height(Lat,Long) + Correct
    enddo
  enddo
  do Lat = 1,n
    print * , (Height(Lat,Long),Long=1,n)
  enddo
end program ch0703
```

The key statement in this example is

```
print * , (Height(Lat,Long),Long=1,n)
```

This is called an implied do loop, as the longitude variable takes on values from 1 through 3 and will write out all three values on one line.

We will see other examples of this statement as we go on.

7.2.3 Example 4: Average of Three Sets of Values

This example extends the previous one. Now we have three sets of measurements and we are interested in calculating the average of these three sets. The two new data sets are:

```
  9.5    39.5    69.5
 19.5    49.5    79.5
 29.5    59.5    89.5
```

and

```
      10.5    40.5    70.5
      20.5    50.5    80.5
      30.5    60.5    90.5
```

and we have chosen the values to enable us to quickly check that the calculations for the averages are correct.

This program also uses implied do loops to read the data, as data in files are generally tabular:

```
program ch0704
! Variables used
! H1,H2,H3 - used to hold the heights above sea level
! H4 - used to hold the average of the above
! Long - used to represent the longitude
! Lat - used to represent the latitude
!    both restricted to integer values.
implicit none
integer , parameter :: n = 3
integer :: Lat , Long
real , dimension(1:n,1:n) :: H1,H2,H3,H4
   do Lat = 1,n
        read * , (H1(Lat,Long), Long=1,n)
   enddo
   do Lat = 1,n
        read * , (H2(Lat,Long), Long=1,n)
   enddo
   do Lat = 1,n
        read * , (H3(Lat,Long), Long=1,n)
   enddo
   do Lat = 1,n
     do Long = 1,n
       H4(Lat,Long)=( H1(Lat,Long)+ H2(Lat,Long)+ &
                      H3(Lat,Long) ) / n
     enddo
   enddo
   do Lat = 1,n
        print * , (H4(Lat,Long),Long=1,n)
   enddo
end program ch0704
```

The original data was accurate to three significant figures. The output from the above has spurious additional accuracy. We will look at how to correct this in the later chapter on output.

7.2.4 Example 5: 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 2-d or 3-d array. Consider the following program extract:

```
program ch0705
implicit none
integer  , parameter  :: NR=5
integer  , parameter  :: NC=10
integer  , parameter  :: NF=3
integer :: Row,Column,Floor
character*1 , dimension(1:NR,1:NC,1:NF) :: Seats=' '
  do   Floor=1,NF
    do   Row=1,NR
       read *,(Seats(Row,Column,Floor),Column=1,NC)
    enddo
  enddo
  print *,' Seat plan is'
  do  Floor=1,NF
    print *,' Floor = ',Floor
    do  Row=1,NR
      print *,(Seats(Row,Column,Floor),Column=1,NC)
    enddo
  enddo
end program ch0705
```

Note here the use of the term parameter in conjunction with the integer declaration. This is called an entity orientated declaration. An alternative to this is an attribute-orientated declaration, e.g.,

```
integer :: NR,NC,NF
parameter :: NR=5,NC=10,NF=3
```

and we will be using the entity-orientated declaration method throughout the rest of the book. This is our recommended method as you only have to look in one place to determine everything that you need to know about an entity.

7.3 Additional Forms of the Dimension Attribute and do Loop Statement

7.3.1 Example 6: Voltage from –20 to +20 Volts

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

```
program ch0706
implicit none
real , dimension(-20:20) :: Current
real :: Resistance
integer :: Voltage
  print *,' type in the resistance'
  read *, Resistance
  do  Voltage = -20,20
    Current(Voltage)=Voltage/Resistance
    print *, Voltage, ' ', Current(Voltage)
  enddo
end program ch0706
```

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.

This form of the dimension attribute

```
dimension(First:Last)
```

is of considerable use when the problem has an effective index which does not start at 1.

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 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.

7.3.2 Example 7: Longitude from −180 to +180

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

```
program ch0707
implicit none
real , dimension(-180:180) :: Time=0
integer :: Degree,Strip
real :: value
  do Degree=-180,165,15
    value=Degree/15.
    do Strip=0,14
      Time(Degree+Strip)=value
    enddo
  enddo
  do Degree=-180,180
    print *,Degree,' ',Time(Degree)
  end do
end program ch0707
```

7.3.3 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.

What changes would you make to the program to accommodate +180? What is the time at −180 and +180?

7.4 The Do Loop and Straight Repetition

7.4.1 Example 8: Table of Liquid Conversion Measurements

Consider the production of a table of liquid measurements. The independent variable is the litre value; the gallon and US gallon are the dependent variables. 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 shows a complete but simple conversion program:

```
program ch0708
implicit none
!
! 1 us gallon = 3.7854118 litres
! 1 uk gallon = 4.545      litres
!
integer :: litre
real     :: gallon,usgallon
   do litre = 1,10
     gallon    = litre / 4.545
     usgallon  = litre / 3.7854118
     print *,litre, ' ',gallon,' ',usgallon
   end do
end program ch0708
```

Note here that the do statement has been used only to control the repetition of a block of statements — there are no arrays at all in this program.

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.

7.4.2 Example 9: Means and Standard Deviations

In the calculation of the mean and standard deviation of a list of numbers, we can 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, whereas 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 ch0709
! Variables used are
!     Mean - for the running mean
!     SSQ  - The running corrected sum of squares
!     X    - Input values for which
! mean and sd required
!     W    - Local work variable
!     SD   - Standard Deviation
!     R    - Another work variable
```

```
implicit none
real :: Mean=0.0,SSQ=0.0,X,W,SD,R
integer :: i,N
  print *,' ENTER THE NUMBER OF READINGS'
  read*,N
  print*,' ENTER THE ',N,' VALUES, ONE PER LINE'
  do i=1,N
    read*,X
    W=X-Mean
    R=I-1
    Mean=(R*Mean+X)/I
    SSQ=SSQ+W*W*R/I
  enddo
  SD=(SSQ/R)**0.5
  print *,' Mean is ',Mean
  print *,' Standard deviation is ',SD
end program ch0709
```

7.5 Summary

Arrays can have up to fifteen dimensions.

Do loops may be nested, but they must not overlap.

The dimension attribute 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.,

```
real , dimension(-123:-10) :: List
real , dimension(0:100,0:100) :: Surface
real , dimension(1:100) :: value
```

The last example could equally be written

```
real , dimension(100) :: value
```

where the first limit is omitted and is given the default value 1. The array LIST would contain 114 values, while Surface would contain 10201.

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

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.
The formal syntax of the block do construct is

```
[ do-construct-name : ] do [label] [ loop-control ]
   [ execution-part-construct ]
[ label ] end-do
```

where the forms of the loop control are

```
[ , ] scalar-variable-name =
scalar-numeric-expression ,
scalar-numeric-expression
[ , scalar-numeric-expression ]
```

and the forms of the end-do are

```
end do [ do-construct-name ]
continue
```

and [] identify optional components of the block do construct. This statement is looked at in much greater depth in Chap. 13.

7.6 Problems

1. Compile and run all the examples in this chapter, except example 5. This is covered separately later.
2. Modify the first example to convert the height in feet to height in metres. The conversion factor is one 1 ft equals 0.305 m.
   Hint: You can either overwrite the height array or introduce a second array.
3. The following are two equations for temperature conversion

$$c = 5/9*(t-32)$$
$$f = 32+9/5*t$$

   Write a complete program where t is an integer do loop variable and loop from −50 to 250. Print out the values of c, t and f on one line. What do you notice about the c and f values?
4. 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.
   Hint: You don't need to use an array here.

5. Write a program to read the following data into a two-dimensional array:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

   Calculate totals for each row and column and produce output similar to that shown below:

$$\begin{matrix} 1 & 2 & 3 & 6 \\ 4 & 5 & 6 & 15 \\ 7 & 8 & 9 & 25 \\ 12 & 15 & 18 & \end{matrix}$$

   Hint 1: Example ch0602 shows how to sum over a loop.

   Hint 2: You need to introduce two one-dimensional arrays to hold the row and column totals. You need to index over the rows to get the column totals and over the columns to get the row totals.
6. Modify the above to produce averages for each row and column as well as the totals.
7. Using the following data from problem 4.1 in Chap. 6:

```
1.85      85
1.80      76
1.85      85
1.70      90
1.75      69
1.67      83
1.55      64
1.63      57
1.79      65
1.78      76
```

   Use the program that evaluated the mean and standard deviation to do so for these heights and weights.

   In the first case use the program as is and run it twice, first with the heights then with the weights.

   What changes would you need to make to the program to read a height and a weight in a pair?

   Hint: You could introduce separate scalar variables for the heights and weights.
8. Example 5 looked at seat bookings in a cinema or theatre. Here is an example of a sample data file for this program

```
P  P  P  P  P  P  P  P  P  P
P  P  P  C  C  C  C  P  P  P
C  C  C  E  E  P  P  P  P  P
C  C  C  C  C  C  C  C  C  C
E  E  E  P  P  P  P  P  P  P
C  C  E  E  P  P  C  C  E  E
P  P  P  P  P  P  P  P  P  P
P  P  P  C  C  C  C  P  P  P
C  C  C  E  E  P  P  P  P  P
C  C  C  C  C  C  C  C  C  C
E  E  E  P  P  P  P  P  P  P
C  C  E  E  P  P  C  C  E  E
P  P  P  P  P  P  P  P  P  P
P  P  P  C  C  C  C  P  P  P
C  C  C  E  E  P  P  P  P  P
```

The key for this is as follows:

C = Confirmed Booking
P = Provisional Booking
E = Seat Empty

Compile and run the program. The output would benefit from adding row and column numbers to the information displayed. We will come back to this issue in a subsequent chapter on output formatting.
The data are in a file on the web and the address is given below.

- http://www.fortranplus.co.uk

Problem 4.3 in the last chapter shows how to read data from a file.

Chapter 8
Whole Array and Additional Array Features

A good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher.

Bertrand Russell

Aims

The aims of the chapter are:

- To look more formally at the terminology required to precisely describe arrays.
- To introduce ways in which we can manipulate whole arrays and parts of arrays (sections).
- To introduce the concept of array element ordering and physical and virtual memory.
- To introduce ways in which we can initialise arrays using array constructors.
- To introduce the where statement and array masking.
- To introduce the forall statement and construct.

8.1 Terminology

Fortran supports an abundance of array handling features. In order to make the description of these features more precise a number of additional terms have to be covered and these are introduced and explained below.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_8, © Springer-Verlag London Limited 2012

8.1.1 Rank

The number of dimensions of an array is called its rank. A one-dimensional array has rank 1, a two-dimensional array has rank 2 and so on.

8.1.2 Bounds

An array's bounds are the upper and lower limits of the index in each dimension.

8.1.3 Extent

The number of elements along a dimension of an array is called the extent.

```
integer, dimension(-10:15):: Current
```

has bounds −10 and 15 and an extent of 26.

8.1.4 Size

The total number of elements in an array is its size.

8.1.5 Shape

The shape of an array is determined by its rank and its extents in each dimension.

8.1.6 Conformable

Two arrays are said to be conformable if they have the same shape, that is, they have the same rank and the same extent in each dimension.

8.1.7 Array Element Ordering

Array element ordering states that the elements of an array, regardless of rank, form a linear sequence. The sequence is such that the subscripts along the first dimension vary most rapidly, and those along the last dimension vary most slowly. This is best illustrated by considering, for example, a rank 2 array A defined by

```
real , dimension(1:4,1:2) :: A
```

A has 8 real elements whose array element order is

A(1,1), A(2,1), A(3,1), A(4,1), A(1,2), A(2,2), A(3,2), A(4,2)

i.e., mathematically by column and not row.

8.2 Whole Array Manipulation

The examples of arrays so far have shown operations on arrays via array elements. One of the significant features of Fortran is its ability to manipulate arrays as whole objects. This allows arrays to be referenced not just as single elements but also as groups of elements. Along with this ability comes a whole host of intrinsic procedures for array processing. These procedures are mentioned in Chap. 12, and listed in alphabetical order with examples in Appendix C.

8.2.1 Assignment

An array name without any indices can appear on both sides of assignment and input and output statements. For example, values can be assigned to all the elements of an array in one statement:

```
real, dimension(1:12):: Rainfall
Rainfall=0.0
```

The elements of one array can be assigned to another:

```
integer, dimension(1:50) :: A,B
…
A=B
```

Arrays A and B must be conformable in order to do this.

The following example is illegal since X is rank 1 and extent 20, whilst Z is rank 1 and extent 41.

```
real, dimension(1:20) :: X
real, dimension(1:41) :: Z
X=50.0
Z=X
```

But the following is legal because both arrays are now conformable, i.e., they are both of rank 1 and extent 41:

```
real , dimension (-20:20) :: X
real , dimension (1:41) :: Y
X=50.0
Y=X
```

8.2.2 Expressions

All the arithmetic operators available to scalars are available to arrays, but care must be taken because mathematically they may not make sense.

```
real , dimension (1:50) :: A,B,C,D,E
C=A+B
```

adds each element of A to the corresponding element of B and assigns the result to C.

```
E=C*D
```

multiplies each element of C by the corresponding element of D. This is not vector multiplication. To perform a vector dot product there is an intrinsic procedure `dot_product`, and an example of this is given in a subsequent section on array constructors.

For higher dimensions

```
real ,dimension (1:10,1:10) :: F,G,H
F=F**0.5
```

takes the square root of every element of F.

```
H=F+G
```

adds each element of F to the corresponding element of G.

```
H=F*G
```

multiplies each element of F by the corresponding element of G. The last statement is not matrix multiplication. An intrinsic procedure `matmul` performs matrix multiplication; further details are given in Appendix C.

8.2.3 Example 1: One Dimensional Whole Arrays in Fortran

Consider the following example, which is a solution to a problem set earlier, but is now addressed using some of the whole array features of Fortran

```
program ch0801
implicit none
integer , parameter :: N=12
real , dimension(1:N) :: RainFall_ins=0.0
real , dimension(1:N) :: RainFall_cms=0.0
integer :: Month
  print *, ' Input the rainfall values in inches'
  read *, RainFall_ins
  RainFall_cms=RainFall_ins * 2.54
  do Month=1,N
    print * , ' ', Month , ' ' , &
                   RainFall_ins(Month) , ' ' , &
                   RainFall_cms(Month)
  end do
end program ch0801
```

The statements

```
real , dimension(1:N) :: RainFall_ins=0.0
real , dimension(1:N) :: RainFall_cms=0.0
```

are examples of whole array initialisation. Each element of the arrays is set to 0.0.

The statement

```
  read *, RainFall_ins
```

is an example of whole array I/O, where we no longer have to use a do loop to read each element in.

Finally, we have the statement

```
  RainFall_cms=RainFall_ins * 2.54
```

which is an example of whole array arithmetic and assignment.

8.2.4 Example 2: Two Dimensional Whole Arrays in Fortran

Here is a two-dimensional example:

```
program ch0802
! This program reads in a grid of temperatures
! (degrees Fahrenheit) at 25 grid references
! and converts them to degrees Celsius
implicit none
integer , parameter :: n=5
real, dimension (1:n,1:n) :: Fahrenheit, Celsius
integer :: Long, Lat
!
! read in the temperatures
!
  do Lat=1,n
    print *, ' For Latitude= ',Lat
    do Long=1,n
      print *, ' For Longitude', Long
      read *,Fahrenheit( Lat,Long)
    end do
  end do
!
! Conversion applied to all values
!
  Celsius = 5.0/9.0 * (Fahrenheit - 32.0)
  print * , Celsius
  print * , Fahrenheit
end program ch0802
```

Note the use of whole arrays in the print statements. The output does look rather messy though, and also illustrates array element ordering.

8.3 Array Sections

Often it is necessary to access part of an array rather than the whole, and this is possible with Fortran's powerful array manipulation features.

8.3.1 Example 3: Rank 1 Array Sections

Consider the following:

```
program ch0803
implicit none
integer , dimension(-5:5) :: x
integer :: i
  x(-5:-1)   = -1
  x(0)        =  0
  x(1:5)     =  1
  do i=-5,5
    print *,' ',I,' ',x(i)
  end do
end program ch0803
```

The statement

```
x(-5:-1)=-1
```

is working with a section of an array. It assigns the value –1 to elements x(–5) through x(–l).

The statement

```
x(1:5)   = 1
```

is also working with an array section. It assigns the value 1 to elements x(1) through x(5).

8.3.2 Example 4: Rank 2 Array Sections

In Chap. 6 we gave an example of a table of examination marks, and this is given again below:

Name	Physics	Maths	Biology	History	English	French
Fowler L.	50	47	28	89	30	46
Barron L.W	37	67	34	65	68	98
Warren J.	25	45	26	48	10	36
Mallory D.	89	56	33	45	30	65
Codd S.	68	78	38	76	98	65

The following program reads the data in, scales column 3 by 2.5 as the Biology marks were out of 40 (the rest are out of 100), calculates the averages for each subject and for each person and prints out the results.

```
program ch0804
implicit none
integer , parameter :: nrow=5
integer , parameter :: ncol=6
real , dimension(1:nrow,1:ncol) &
                      :: Exam_Results      = 0.0
real , dimension(1:nrow) &
                      :: People_average    = 0.0
real , dimension(1:ncol) &
                      :: Subject_Average = 0.0
integer :: r,c
  do r=1,nrow
    read *, exam_results(r,1:ncol)
  end do
  Exam_Results(1:nrow,3) = 2.5 * Exam_Results(1:nrow,3)
  do r=1,nrow
    do c=1,ncol
      people_average(r) = people_average(r) + &
                          exam_results(r,c)
    end do
  end do
  people_average = people_average / ncol
  do c=1,ncol
    do r=1,nrow
      subject_average(c) = subject_average(c) + &
                           exam_results(r,c)
    end do
  end do
  subject_average = subject_average / nrow
  print *,' People averages'
  print *, people_average
  print *, ' Subject averages'
  print *, subject_average
end program ch0804
```

The statement

```
read *, exam_results(r, 1:ncol)
```

uses sections to replace the implied do loop in the earlier example.

The statement

```
Exam_Results(1:nrow,3) = 2.5 * Exam_Results(1:nrow,3)
```

uses array sections in the arithmetic and the assignment.

8.4 Array Constructors

Arrays can be given initial values in Fortran using array constructors. Some examples are given below.

8.4.1 Example 5: Rank 1 Array Initialisation – Explicit Values

```
program ch0805
implicit none
integer , parameter :: n=12
real :: Total=0.0, Average=0.0
real , dimension(1:n) :: RainFall = &
  (/3.1,2.0,2.4,2.1,2.2,2.2,1.8,2.2,2.7,2.9,3.1,3.1/)
integer :: Month
  do Month=1,n
    Total = Total + RainFall(Month)
  enddo
  Average = Total / n
  print *,' Average monthly rainfall was'
  print *, Average
end program ch0805
```

The statement

```
real , dimension(1:n) :: RainFall = &
  (/3.1,2.0,2.4,2.1,2.2,2.2,1.8,2.2,2.7,2.9,3.1,3.1/)
```

provides initial values to the elements of the array Rainfall.

8.4.2 Example 6: Rank 1 Array Initialisation Using an Implied do Loop

The next example uses a simple variant:

```
program ch0806
implicit none
!
! 1 us gallon = 3.7854118 litres
! 1 uk gallon = 4.545      litres
!
integer , parameter :: n=10
real , parameter :: us = 3.7854118
real , parameter :: uk = 4.545
integer :: i
integer , dimension(1:n) :: Litre=[(i,i=1,n)]
real    , dimension(1:n) :: Gallon,USGallon
  Gallon    = Litre / uk
  USGallon = Litre / us
  print *,'  Litres           Imperial        USA'
  print *,'                    Gallon          Gallon'
  do i = 1,n
    print *,Litre(i), ' ',Gallon(i),' ',USGallon(i)
  end do
end program ch0806
```

The statement

```
integer , dimension(1:n) :: Litre=[(i,i=1,n)]
```

initialises the 10 elements of the Litre array to the values 1,2,3,4,5,6,7,8,9,10 respectively.

8.4.3 Example 7: Rank 1 Arrays and the `dot_product` Intrinsic

The following example uses an array constructor and the intrinsic procedure `dot_product`:

```
program ch0807
implicit none
integer , dimension(1:3) :: X,Y
integer :: result
 X=[1,3,5]
 Y=[2,4,6]
 result=dot_product(X,Y)
 print *,result

end program ch0807
```

and result has the value 44, which is obtained by the normal mathematical dot product operation, 1*2+3*4+5*6.

The general form of the array constructor is [list of expressions] or (/ a list of expressions/) where each expression is of the same type.

8.5 Initialising Rank 2 Arrays

To construct arrays of higher rank than one the intrinsic function RESHAPE must be used. An introduction to intrinsic functions is given in Chap. 12, and an alphabetic list with a full explanation of each function is given in Appendix C. To use it in its simplest form:

```
Matrix = reshape ( Source, Shape)
```

where Source is a rank 1 array containing the values of the elements required in the new array, Matrix, and Shape is a rank 1 array containing the shape of the new array Matrix.

We consider the rank 1 array B = (1,3,5,7,9,11), and we wish to store these values in a rank 2 array A, such that A is the matrix:

$$A = \begin{pmatrix} 1 & 7 \\ 3 & 9 \\ 5 & 11 \end{pmatrix}$$

The following code extract is needed:

```
integer, dimension(1:6) :: B
integer, dimension(1:3, 1:2) :: A
  B = (/1,3,5,7,9,11/)
  A = reshape(B,(/3,2/))
```

Note that the elements of the source array B must be stored in the array element order of the required array A.

8.5.1 Example 8: Initialising a Two Dimensional Array

The following example illustrates the additional forms of the `reshape` function that are used when the number of elements in the source array is less than the number of elements in the destination. The complete form is

```
reshape (source, shape, pad, order)
```

`Pad` and `Order` are optional. See Appendix C for a complete explanation of Pad and Order:

```
program ch0808
implicit none
integer , dimension(1:2,1:4) :: x
integer , dimension(1:8)   :: y=(/1,2,3,4,5,6,7,8/)
integer , dimension(1:6)   :: z=(/1,2,3,4,5,6/)
integer :: r,c
  print *,' Source array y'
  print *,y
  print *,' Source array z'
  print *,z
  print *,' Simple reshape sizes match'
  x=reshape(y,(/2,4/))
  do r=1,2
    print *,(x(r,c),c=1,4)
  end do
  print *,' Source 2 elements smaller pad with 0'
  x=reshape(z,(/2,4/),(/0,0/))
  do r=1,2
    print *,(x(r,c),c=1,4)
  end do
  print *,' As previous now specify order as 1*2'
  x=reshape(z,(/2,4/),(/0,0/),(/1,2/))
  do r=1,2
    print *,(x(r,c),c=1,4)
  end do
  print *,' As previous now specify order as 2*1'
  x=reshape(z,(/2,4/),(/0,0/),(/2,1/))
  do r=1,2
    print *,(x(r,c),c=1,4)
  end do
end program ch0808
```

8.6 Miscellaneous Array Examples

The following are examples of some of the flexibility of arrays in Fortran.

8.6.1 Example 9: Rank 1 Arrays and a Step Size of 2 in Implied Do Loop

Consider the following example:

```
program ch0809
implicit none
integer :: i
integer , dimension(1:10) :: x=(/(i,i=1,10)/)
integer , dimension(1:5)  :: odd=(/(i,i=1,10,2)/)
integer , dimension(1:5)  :: even
  even=x(2:10:2)
  print *,' x'
  print *,x
  print *,' odd'
  print *,odd
  print *,' even'
  print *,even
end program ch0809
```

The statement

```
integer , dimension(1:5) :: odd=(/(i,i=1,10,2)/)
```

steps through the array 2 at a time.

The statement

```
  even=x(2:10:2)
```

shows an array section where we go from elements two through ten in steps of two. The 2:10:2 is an example of a subscript triplet in Fortran, and the first 2 is the lower bound, the 10 is the upper bound, and the last 2 is the increment. Fortran uses the term stride to mean the increment in a subscript triplet.

8.6.2 Example 10: Rank 1 Array and the `sum` Intrinsic Function

The following example is based on ch0805. It uses the sum intrinsic to calculate the sum of all the values in the Rainfall array.

```
program ch0810
implicit none
real :: Total=0.0, Average=0.0
real , dimension(12) :: RainFall = &
  (/3.1,2.0,2.4,2.1,2.2,2.2,1.8,2.2,2.7,2.9,3.1,3.1/)
integer :: Month
```

```
  Total = sum(RainFall)
  Average = Total / 12
  print *,' Average monthly rainfall was'
  print *, Average
end program ch0810
```

The statement

```
  Total = sum(RainFall)
```

replaces the statements below from the earlier example

```
  do Month=1,n
    Total = Total + RainFall(Month)
  enddo
```

In this example `sum` adds up all of the elements of the array Rainfall.

So we have three ways of processing arrays:

- Element by element.
- Using sections.
- On a whole array basis.

The ability to use sections and whole arrays when programming is a major advance of the element by element processing supported by Fortran 77.

8.6.3 *Example 11: Rank 2 Arrays and the* `sum` *Intrinsic Function*

This example is based on the earlier exam results program:

```
program ch0811
implicit none
integer , parameter :: nrow=5
integer , parameter :: ncol=6
real , dimension(1:nrow*ncol)  :: results = &
                (/50 , 47 , 28 , 89 , 30 , 46 , &
                  37 , 67 , 34 , 65 , 68 , 98 , &
                  25 , 45 , 26 , 48 , 10 , 36 , &
                  89 , 56 , 33 , 45 , 30 , 65 , &
                  68 , 78 , 38 , 76 , 98 , 65/)
real , dimension(1:nrow,1:ncol) :: Exam_Results &
    = 0.0
real , dimension(1:nrow)        :: People_average &
    = 0.0
real , dimension(1:ncol)         :: Subject_Average &
    = 0.0
```

```
integer :: r,c
  exam_results = &
    reshape(results,(/nrow,ncol/),(/0.0,0.0/),(/2,1/))
  Exam_Results(1:nrow,3) = 2.5 * Exam_Results(1:nrow,3)
  subject_average = sum(exam_results,dim=1)
  people_average  = sum(exam_results,dim=2)
  people_average  = people_average / ncol
  subject_average = subject_average / nrow
  print *,' People averages'
  print *, people_average
  print *, ' Subject averages'
  print *, subject_average
end program ch0811
```

This example has several interesting array features:

- We initialise a rank 1 array with the values we want in our exam marks array. The data are laid out in the program as they would be in an external file in rows and columns.
- We use `reshape` to initialise our exam marks array. We use the fourth parameter (/2,1/) to populate the rank 2 array with the data in row order.
- We use `sum` with a `dim` of 1 to compute the sums for the subjects.
- We use `sum` with a `dim` of 2 to compute the sums for the people.

8.6.4 Example 12: Masked Array Assignment and the Where Statement

Fortran has array assignment both on an element by element basis and on a whole array basis. There is an additional form of assignment based on the concept of a logical mask.

Consider the example of time zones given in Chap. 7. The `Time` array will have values that are both negative and positive. We can then associate the positive values with the concept of east of the Greenwich meridian, and the negative values with the concept of west of the Greenwich meridian e.g.:

```
program ch0812
implicit none
real , dimension(-180:180) :: Time=0
integer :: Degree,Strip
real :: value
character (len=1) , dimension(-180:180) &
    :: Direction=' '
  do Degree=-180,165,15
    value=Degree/15.
    do Strip=0,14
      Time(Degree+Strip)=value
```

```
    enddo
  enddo
  do Degree=-180,180
    print *,Degree,' ',Time(Degree)
  end do
  where (Time > 0.0)
    Direction='E'
  elsewhere (Time < 0.0)
    Direction='W'
  endwhere
  print *,direction
end program ch0812
```

8.6.5 Notes

The arrays must be conformable, i.e., in our example Time and Direction are the same shape.

The selective assignment is achieved through the where statement.

Both the where and elsewhere blocks can be executed.

The formal syntax is:

```
where (array logical assignment)
   array assignment block
elsewhere
   array assignment block
end where
```

The first array assignment is executed where Time is positive and the is executed where Time is negative. For further coverage of logical expressions see Chaps. 15 and 18.

8.7 The `forall` Statement and `forall` Construct

The `forall` statement and `forall` construct were introduced into Fortran to keep it inline with High Performance Fortran – HPF. They indicate to the compiler that the code can be optimised on a parallel processor. Consider the following example where a value is subtracted from the diagonal elements of a square matrix A:

```
forall (I=1:N)
  A(I,I)=A(I,I) - Lamda
end forall
```

The forall construct allows the calculations to be carried out simultaneously in a multiprocessor environment.

8.7.1 Syntax

```
forall ( triplet  [ , triplet ] ... [ , mask ] )
variable = expression
forall ( triplet  [ , triplet ] ... [ , mask ] )
pointer => target
```

The triplet specifies a value set for an index variable. It has the following syntax:

```
index = first : last [ : stride ]
```

First, last and stride are scalar integer expressions.

Mask is a scalar logical expression:

```
[ name : ] forall ( triplet [ , triplet ] ... [ , mask
] )
...
end forall [ name ]
```

Name is an optional name, which identifies the `forall` construct.

8.7.2 Array Element Ordering and Physical and Virtual Memory

Fortran compilers will store arrays in memory according to the array element ordering scheme. Whilst the standard says nothing about how this is implemented it generally means in contiguous memory locations.

There will be a limit to the amount of physical memory available on any computer system. To enable problems that require more than the amount of physical memory available to be solved, most implementations will provide access to virtual memory, which in reality means access to a portion of a physical disk.

Access to virtual memory is commonly provided by a paging mechanism of some description. Paging is a technique whereby fixed-sized blocks of data are swapped between real memory and disk as required.

In order to minimise paging (and hence reduce execution time) array operations should be performed according to the array element order.

Page sizes, past and present, include:

- Sun UltraSparc – 4Kb, 8Kb.
- DEC Alpha – 8Kb, 16Kb, 32Kb, 64Kb.

- Intel 80x86 – 4Kb.
- Intel Pentium PIII – 4Kb, 2Mb, 4Mb.
- AMD64 – 4Kb, 2Mb, 4Mb – legacy mode
- AMD64 – 4Kb, 2Mb, 1Gb – 64 bit mode
- Intel 64 and IA – 32 – 4Kb, 2Mb, 1Gb – depending on mode.

See the references at the end of the chapter for more details.

8.8 Summary

We can now perform operations on whole arrays and partial arrays (array sections) without having to refer to individual elements. This shortens program development time and greatly clarifies the meaning of programs.

Array constructors can be used to assign values to rank 1 arrays within a program unit. The `reshape` function allows us to assign values to a two or higher rank array when used in conjunction with an array constructor.

We have introduced the concept of a deferred-shape array. Arrays do not need to have their shape specified at compile time, only their rank. Their actual shape is deferred until runtime. We achieve this by the combined use of the allocatable attribute on the variable declaration and the allocate statement, which makes Fortran a very flexible language for array manipulation.

8.9 Problems

1. Give the rank, bounds, extent and size of the following arrays:

```
real , dimension(1:15) :: A
integer , dimension(1:3,0:4) :: B
real , dimension(-2:2,0:1,1:4) :: C
integer , dimension(0:2,1:5) :: D
```

   Which two of these arrays are conformable?
2. Write a program to read in five rank 1 arrays, A, B, C, D, E and then store them as five columns in a rank 2 array TABLE.
3. Take the first part of Problem 5 in Chap. 7 and rewrite it using the sum intrinsic function.

8.10 Bibliography

Bhandarkar, D.P.: Alpha Implementation and Architecture: Complete Reference and Guide. Digital Press, Newton (1996)

Amd
Visit

http://developer.amd.com/documentation/guides/pages/default.aspx

for details of the AMD manuals. The following five manuals are available for download as pdf s from the above site.

- AMD64 Architecture Programmer's Manual Volume 1: Application Pro gramming
- AMD64 Architecture Programmer's Manual Volume 2: System Program ming
- AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions
- AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256 bit media instructions
- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions

Intel
Visit

http://www.intel.com/products/processor/manuals/index.htm

for a list of manuals. The following three manuals are available for download as pdf's from the above site.

- Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture
- Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 2A and 2B: Instruction Set Reference, A-Z.
- Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2

Chapter 9
Output of Results

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

Lewis Carroll, Through the Looking-Glass
and What Alice Found There

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.

9.1 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 are 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 are 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.
- Specify where a decimal point should lie.
- Specify where there should be white space.
- Specify titles.

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 them.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_9, © Springer-Verlag London Limited 2012

9.2 Example 1: Integers – I Format or Edit Descriptor

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. Consider the following example:

```
program ch0901
implicit none
integer :: T
  print *,' '
  print *,' Twelve times table'
  print *,' '
  do T=1,12
    print 100, T,T*12
    100 format(' ',I3,' * 12 = ',I3)
  end do
end program ch0901
```

The first statement of interest is

```
print 100, T,T*12
```

The 100 is a statement label. There must be a format statement with this label in the program. The variables to be written out are T and 12*T.

The second statement of interest is

```
100 format(' ',I3,' * 12 = ',I3)
```

Inside the brackets we have

`''`	print out what occurs between the quote marks, in this case one space.
,	The comma separates items in the format statement.
I3	print out the first variable in the print statement right justified in three columns
,	Item separator.
`' * 12 = '`	print out what occurs between the quote characters.
,	Item separator
I3	print out the second variable (in this case an expression) right justified in three columns.

All of the output will appear on one line.

9.3 Example 2: The x Edit Descriptor

Now consider the following example:

```
program ch0902
implicit none
integer :: big=10
integer :: i
  do i=1,40
    print 100,i,big
    100 format(1x,i3,2x,i12)
    big=big*10
  end do
end program ch0902
```

The new feature in the format statement is the 1x and 2x edit descriptor. This is another way of getting white space into the output, and in this case one space and two spaces, respectively.

This program will loop and the variable big will overflow, i.e., go beyond the range of valid values for a 32-bit integer. Does the program crash or generate a run time error? This is the output from the NAG and Intel compilers.

```
 1                   10
 2                  100
 3                 1000
 4                10000
 5               100000
 6              1000000
 7             10000000
 8            100000000
 9           1000000000
10           1410065408
11           1215752192
12           -727379968
13           1316134912
14            276447232
15          -1530494976
16           1874919424
17           1569325056
18          -1486618624
19          -1981284352
20           1661992960
21           -559939584
22          -1304428544
```

(continued)

(continued)

23	-159383552
24	-1593835520
25	1241513984
26	-469762048
27	-402653184
28	268435456
29	-1610612736
30	1073741824
31	-2147483648
32	0
33	0
34	0
35	0
36	0
37	0
38	0
39	0
40	0

Is there a compiler switch to trap this kind of error?

9.4 Reals – F Format or Edit Descriptor

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:

- There is a total width of seven.
- There is a decimal point
- There are four digits after the decimal point.

This means that since the decimal point is also written out, there may be up to two 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
```

The program in Section 9.4.1 illustrates the use of both I format and F format.

9.4.1 Example 3: Metric and Imperial Conversion and the f Edit Descriptor

```
program ch0903
implicit none
integer :: fluid
real :: litres
real :: pints
  do fluid=1,10
    litres = fluid / 1.75
    pints = fluid * 1.75
    print 100 , pints,fluid,litres
    100 format(' ',F7.3,' ',I3,' ',F7.3)
  end do
end program ch0903
```

Pints will be printed out in F7.3 format, fluid will be printed out in I3 format and litres will be printed out in F7.3 format.

9.4.2 Example 4: Overflow and Underflow and the f Edit Descriptor

Consider the following program:

```
program ch0904
implicit none
integer :: i
real     :: small = 1.0
real     :: big   = 1.0
  do i=1,50
    print 100,i,small,big
    100 format(' ',i3,' ',f7.3,' ',f7.3)
    small=small/10.0
    big=big*10.0
  end do
end program ch0904
```

In this program the variable small will underflow and big will overflow. The output from the Intel compiler is:

```
    1   1.000   1.000
    2   0.100  10.000
    3   0.010 100.000
    4   0.001 *******
    5   0.000 *******
    6   0.000 *******
    7   0.000 *******
    8   0.000 *******
...
...
   37   0.000 *******
   38   0.000 *******
   39   0.000 *******
   40   0.000  Infini
   41   0.000  Infini
   42   0.000  Infini
...
   48   0.000  Infini
   49   0.000  Infini
   50   0.000  Infini
```

When the number is too small for the format, the printout is what you would probably expect. When the number is too large, you get asterisks. When the number actually overflows the Intel compiler tells you that the number is too big and has overflowed. However the program ran to completion and did not generate a run time error.

9.5 Reals – E Format or Edit Descriptor

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, 1,000 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.
- 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 0–99, 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 four are valid E formats on output:

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

but the next five would not be acceptable as output formats, for a variety of reasons:

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

9.5.1 Example 5: Simple e Edit Descriptor Usage

This is the same as ch0904 except that we have replaced the F formatting with E formatting:

```
program ch0905
implicit none
integer :: i
real    :: small = 1.0
real    :: big   = 1.0
  do i=1,50
    print 100,i,small,big
    100 format(' ',i3,' ',e10.4,' ',e10.4)
    small=small/10.0
    big=big*10.0
  end do
end program ch0905
```

We now have three ways to print out floating point numbers and each has its use. The print * is very useful when developing programs.

9.6 Spaces

You have seen two ways of generating spaces on output. The first is to use ' characters to enclose blanks in the format statement. The second is to use the X edit descriptor. Consider the following.

```
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.

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 a positive integer followed by X.

9.7 Characters – A Format or Edit Descriptor

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(1X,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 10 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.

9.7.1 Example 6: Character Output and the a Edit Descriptor

The following program is a simple rewrite of a program from Chap. 4.

```
program ch0906
!
! This program reads in and prints out
! your first name
!
implicit none
character (20) :: first_name
!
  print *,' type in your first name.'
  print *,' up to 20 characters'
  read *,first_name
  print 100,first_name
  100 format(1x,A)
!
end program ch0906
```

9.7.2 Example 7: Headings

A simple heading is given in the program below:

```
program ch0907
implicit none
integer :: fluid
real :: litres
real :: pints
  print *,' Pints       Litres'
  do fluid=1,10
    litres = fluid / 1.75
    pints  = fluid * 1.75
    print 100 , pints,fluid,litres
    100 format(' ',f7.3,' ',i3,' ',f7.3)
  end do
end program ch0907
```

9.8 Example 8: Mixed Type Output in a Format Statement

The following example shows how to mix and match character, integer and real output in one format statement:

```
program ch0908
implicit none
character (len=15) :: Firstname
integer :: age
real :: weight
character (len=1) :: sex
  print *,' type in your first name '
  read *,Firstname
  print *,' type in your age in years'
  read *,age
  print *,' type in your weight in kilos'
  read *,weight
  print *,' type in your sex (f/m)'
  read *,sex
  print *,' your personal details are'
  print *
  print 100
  100 format(4x,'first name', 4x , 'age' , 1x , &
     'weight' , 2x , 'sex')
  print 200 , firstname, age , weight , sex
  200 format (1x , a , 2x , i3 , 2x , f5.2 , 2x, a)
end program ch0908
```

Take care to match up the variables with the appropriate edit descriptors. You also need to count the number of characters and spaces when lining up the heading.

9.9 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 use integer variables or expressions with F or E edit descriptors or real variables and expressions with I edit descriptors. if you do, unpredictable results will follow. There are (at least) two other sorts of errors you might make in writing out a value. You might try to write out something which has never actually been assigned a value; this is termed an indefinite value. You might 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 mistyping rather than of 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 the machine can calculate – out of range values. Some machines will print out such values as R, but some will actually print out something which looks right, and such overflow and underflow conditions can go unnoticed. Be wary.

9.10 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 also possible to read and write to one or more files. This is achieved using the open, write, read and close statements. In a later chapter we will consider reading from files but here we will concentrate on writing.

9.10.1 The Open Statement

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

```
open (unit=1,file='data.txt')
```

The file will be known to the operating system as data.txt (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.txt directly; it must be referenced through its unit number. Within the Fortran program you write to this file using a statement such as

```
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.

9.10.2 Example 9: Open and Close Usage

The following program contains all of the above statements:

```
program ch0909
implicit none
integer :: fluid
real :: litres
real :: pints
  open (unit=1,file='ch0909.txt')
  write(unit=1,fmt=200)
  200 format(' Pints        Litres')
  do fluid=1,10
    litres = fluid / 1.75
    pints = fluid * 1.75
    write(unit=1,fmt=100) pints,fluid,litres
    100 format(' ',f7.3,' ',i3,' ',f7.3)
  end do
  close(1)
end program ch0909
```

9.10.3 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.

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 data. 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)
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.

9.11 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 microcomputers and workstations). 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 a subdivision within a file. Even then, there was some confusion between the notion of physical records and files split into logically distinct subunits, 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 and 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.

9.12 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 all seems 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 numbers of data, and neither uses up the whole format.

9.13 Example 10: Implied Do Loops and Array Sections for Array Output

The following program shows how to use both implied do loops and array sections to output an array in a neat fashion:

```
program ch0910
implicit none
integer , parameter :: nrow=5
integer , parameter :: ncol=6
real , dimension(1:nrow*ncol)  :: results = &
                (/50 , 47 , 28 , 89 , 30 , 46 , &
                  37 , 67 , 34 , 65 , 68 , 98 , &
                  25 , 45 , 26 , 48 , 10 , 36 , &
                  89 , 56 , 33 , 45 , 30 , 65 , &
                  68 , 78 , 38 , 76 , 98 , 65/)
real , dimension(1:nrow,1:ncol) :: &
  Exam_Results     = 0.0
real , dimension(1:nrow)         :: &
  People_average = 0.0
real , dimension(1:ncol)         :: &
  Subject_Average = 0.0
integer :: r,c
  exam_results = &
  reshape(results,(/nrow,ncol/),(/0.0,0.0/),(/2,1/))
```

```
  Exam_Results(1:nrow,3) = 2.5 * Exam_Results(1:nrow,3)
  subject_average = sum(exam_results,dim=1)
  people_average  = sum(exam_results,dim=2)
  people_average  = people_average / ncol
  subject_average = subject_average / nrow
  do r=1,nrow
    print 100 , (exam_results(r,c),c=1,ncol) ,&
      people_average(r)
    100 format(1x,6(1x,f5.1),' = ',f6.2)
  end do
  print *,'  ====  ====  ====  ====  ====  ===='
  print 110, subject_average(1:ncol)
  110 format(1x,6(1x,f5.1))
end program ch0910
```

The print 100 uses an implied do loop and the print 110 uses an array section.

9.13.1 Example 11: Whole Array Output

Take care when using whole arrays. Consider the following program:

```
program ch0911
real , dimension(10,10) :: Y
integer :: NROWS=6
integer :: NCOLS=7
integer :: i,j
integer :: K=0

  do i=1,NROWS
    do J=1,NCOLS
      K=K+1
      Y(I,J)=K
    end do
  end do

  write(unit=*,fmt=100)Y
  100 format(1X,10F10.4)

end program ch0911
```

There are several points to note with this example. Firstly, this is a whole array reference, and so 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 is according to Fortran's array element ordering, i.e., the first subscript varying 1–10

(the array bound), with the second subscript as 1, then 1–10 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)
```

Thirdly we have defined values for part of the array.

9.14 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 an effect on standard line printers 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 are 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 overprinted 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 6 in. stack of paper which is (almost) blank, because you had a 1 repeatedly in column 1.

9.14.1 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 four 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.

9.14.2 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 1 of your line printer output. A more general approach, which works on both terminals and 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 it opens up lots of possibilities for formatting output more tidily:

```
    print 102,NVAL,XMAX,XMIN
    102 format(' NUMBER OF VALUES read IN WAS: ',I10/ &
           ' MAXIMUM value IS: ',F10.4/            &
           ' MINIMUM value IS: ',F10.4)
```

may be easier to read than using only one line, and it 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.

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 four 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.

9.15 Example 12: Timing of Writing Formatted Files

The following example looks at the amount of time spent in different sections of a program with the main emphasis on formatted output:

```
program ch0912
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x=0
real    , dimension(1:n)  :: y=0.0
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
  open(unit=10,file='ch0912.dat')
  call cpu_time(t)
```

```
  t1=t
  comment=' Initial '
  print 100,comment,t1
  do i=1,n
    x(i)=I
  end do
  call cpu_time(t)
  t2=t-t1
  comment = ' integer '
  print 100,comment,t2
  y=real(x)
  call cpu_time(t)
  t3=t-t1-t2
  comment = ' real '
  print 100,comment,t2
  do i=1,n
    write(10,200) x(i)
    200 Format (1x,i10)
  end do
  call cpu_time(t)
  t4=t-t1-t2-t3
  comment = ' I write '
  print 100,comment,t4
  do i=1,n
    write(10,300) y(i)
    300 Format (1x,f10.0)
  end do
  call cpu_time(t)
  t5=t-t1-t2-t3-t4
  comment = ' r write '
  print 100,comment,t5
  100 format(1x,a,2x,f7.3)
end program ch0912
```

There is a call to the built-in intrinsic cpu_time to obtain timing information. Try this example out with your compiler. Formatted output takes up a lot of time, as we are converting from an internal binary representation to an external decimal form.

9.16 Example 13: Timing of Writing Unformatted Files

The following program is a variant of the above but now the output is in unformatted or binary form:

```
program ch0913
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x=0
real   , dimension(1:n) :: y=0
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
  open(unit=10,file='ch0913.dat',form='unformatted')
  call cpu_time(t)
  t1=t
  comment=' Initial '
  print 100,comment,t1
  do i=1,n
    x(i)=I
  end do
  call cpu_time(t)
  t2=t-t1
  comment = ' integer '
  print 100,comment,t2
  y=real(x)
  call cpu_time(t)
  t3=t-t1-t2
  comment = ' real '
  print 100,comment,t2
  write(10) x
  call cpu_time(t)
  t4=t-t1-t2-t3
  comment = ' I write '
  print 100,comment,t4
  write(10) y
  call cpu_time(t)
  t5=t-t1-t2-t3-t4
  comment = ' r write '
  print 100,comment,t5
  100 format(1x,a,2x,f7.3)
end program ch0913
```

Try this example out with your compiler. Unformatted is very efficient in terms of time. It also has the benefit for real or floating point numbers of no information loss.

9.17 Summary

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

The main features are:

- The I format for integer variables.
- The E and F formats for real numbers.
- The A format for characters.
- The X, which allows insertion of spaces.

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.

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, the following four characters:

- +
- 0
- 1
- (blank)

allow reasonable control over the layout. Care must to be taken with these characters, since it is possible to decimate forests with little effort.

9.18 Problems

1. Rewrite the temperature conversion program which was problem 3 in chapter 7 to produce neat tabula output. Pay attention to the number of significant decimal places.
2. Write a litres and pints conversion program to produce a similar kind of output to problem one above. Start at 0 and make the central column go up to 50. One pint is 0.568 l.
3. Information on car fuel consumption is usually given in miles per gallon in Britain and the United States 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. The two most commonly used operating systems for Fortran programming are UNIX and DOS. It is possible to use the operating system file redirection symbols < and > to read from a file and write to a file, respectively. Rerun the program in problem 1 to write to a file. Examine the file using an editor.
5. 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?
6. To demonstrate your familiarity with formats, reformat problems 1, 2 or 3 to use E formats, rather than F (or vice versa).
7. 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.
8. Repeat for the litres and pints program.
9. What features of Fortran reveal its evolution from punched card input?
10. 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.
11. Check what a number too large for the output format will be printed as on your local system – is it all asterisks?
12. 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.
13. Fortran is an old programming language and the text formatting functionality discussed in this chapter assumes very dumb printing devices.

The primary assumption is that we are dealing with so-called monospace fonts, i.e., that digits, alphabetic characters, punctuation, etc., all have the same width.

If you are using a PC try using:

- Notepad

and

- Word

to open your programs and some of the files created in this chapter. What happens to the layout?

If you are using Notepad look at the Word wrap and set Font options under the edit menu.

What fonts are available? What happens to the layout when you choose another font?

If you are using Word what fonts are available? What happens when you make changes to your file and exit Word? Is it sensible to save a Fortran source file as a Word document?

Chapter 10
Reading in Data

Winnie-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, Winnie-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 zero?
- read – extensions.
  - error handling on input.
- open – associating unit numbers and file names.
  - close
  - rewind
  - backspace

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_10, © Springer-Verlag London Limited 2012

10.1 Reading from the Terminal or Keyboard Versus Reading from Files

It is unlikely that you would use fixed formats when reading numeric input from the terminal or keyboard; they are more likely to be used when reading data from a file. However the examples that follow do it. We look at reading from files later in this chapter.

10.2 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.

10.2.1 Integers and the I Format

Integers are read in with the I edit descriptor. Whereas, 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 0–9, the only other characters which may appear in an integer field are – and +.

10.2.2 Example 1: Skipping Data Whilst Reading

Consider the following 12 times table:

```
 1 * 12 =  12
 2 * 12 =  24
 3 * 12 =  36
 4 * 12 =  48
 5 * 12 =  60
 6 * 12 =  72
 7 * 12 =  84
 8 * 12 =  96
 9 * 12 = 108
10 * 12 = 120
11 * 12 = 132
12 * 12 = 144
```

The following is a program to read the first and last columns of integer data:

```
program ch1001
implicit none
integer , parameter :: n=12
integer :: i
integer , dimension(1:n) :: x
integer , dimension(1:n) :: y
  do i=1,n
     read 100,x(i),y(i)
     100 format (2x,i2,9x,i3)
     print 200,x(i),y(i)
     200 format(1x,i3,2x,i3)
  end do
end program ch1001
```

The

```
read 100,x(i),y(i)
```

will try reading values into x(i) and y(i) using format statement

```
100 format (2x,i2,9x,i3)
```

which will skip the first two characters on the line or record, read the first value from the next two columns, skip the next nine characters and read the last value from the next three characters.

We recommend that when working with formatted files you to use a text editor that displays the column and line details.

Notepad under Windows has a status bar option under the View menu. Gvim under Windows has line and column information available. Under Redhat, vim and gedit both display line and column information. User SuSe, kedit and vim display line and column information. There should be an editor available on your system that has this option.

10.2.3 Reals and the F Format

Real numbers may be input using a variety of formats and we will look at the F format in this example. Consider the following BMI data:

```
1.85  85
1.80  76
1.85  85
1.70  90
1.75  69
1.67  83
1.55  64
```

```
1.63  57
1.79  65
1.78  76
```

The following program will read in the data:

```
program ch1002
implicit none
integer , parameter :: n=10
real , dimension(1:n) :: h
real , dimension(1:n) :: w
real , dimension(1:n) :: bmi
integer :: i
  do i=1,n
    read 100, h(i),w(i)
    100 format(f4.2,2x,f3.0)
  end do
  bmi=w/(h*h)
  do i=1,n
    print 200,bmi(i)
    200 format(2x,f5.0)
  end do
end program ch1002
```

To read in the heights we need a total width of four columns with two after the decimal point. We then skip two spaces and read in the weights. The data in the file do not have a decimal point!

10.2.4 Reals and the E Format

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, so:

```
-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 giving only

the significant digits. If the exponent is to be given, there must be some significant digits as well. 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
```

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 might generate confusion as well. 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. So given the data:

```
136
136
136
136
136.
136.
136.
136.
```

and the program

```
program ch1003
implicit none
real     :: x
  read 100,x
  100 format(e10.0)
  print *,x
  read 200,x
  200 format(e10.4)
  print *,x
  read 300,x
  300 format(e10.10)
  print *,x
  read *,x
  print *,x
  read 100,x
  print *,x
  read 200,x
  print *,x
  read 300,x
  print *,x
  read *,x
  print *,x
end program ch1003
```

We get the following output when the program is compiled with the Intel compiler:

```
 136.0000
1.3600000E-02
1.3600000E-08
 136.0000
 136.0000
 136.0000
 136.0000
 136.0000
```

Other compilers may give slightly different formatting of the output.

10.3 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 were 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.

10.4 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 nine 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.

A simple variant on ch1001 which uses the character variable temp to hold the text between the two numbers appears below:

```
program ch1004
implicit none
integer , parameter :: n=12
integer :: i
integer , dimension(1:n) :: x
integer , dimension(1:n) :: y
character*9 :: temp
  do i=1,n
    read 100,x(i),temp,y(i)
    100 format(2x,i2,a,i3)
    print 200,x(i),y(i)
    200 format(1x,i3,2x,i3)
  end do
end program ch1004
```

Note that in the format statement we just use the A edit descriptor and the number of characters to read is picked up from the variable declaration.

10.5 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=*,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=*,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=*,fmt=100)
100 format()
```

with no variable name at all.

10.6 Reading

As you have already seen, 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=*,fmt=100) X,Y
```

for formatted input 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
```

10.7 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='Example.txt')
read(unit=10,fmt=100) A,B
…
open(unit=10,file='Example.txt',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 is better done by manipulating individual formats if it has 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 later in 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 if the input is from the keyboard or terminal.

10.8 Reading Using Array Sections

Consider the following output:

```
 50.0  47.0  70.0  89.0  30.0  46.0  =  55.33
 37.0  67.0  85.0  65.0  68.0  98.0  =  70.00
 25.0  45.0  65.0  48.0  10.0  36.0  =  38.17
 89.0  56.0  82.5  45.0  30.0  65.0  =  61.25
 68.0  78.0  95.0  76.0  98.0  65.0  =  80.00
 ====  ====  ====  ====  ====  ====
 53.8  58.6  79.5  64.6  47.2  62.0
```

A program to read this file using array sections is as follows:

```
program ch1005
implicit none
integer , parameter :: nrow=5
integer , parameter :: ncol=6
real , dimension(1:nrow,1:ncol) :: &
  Exam_Results     = 0.0
real , dimension(1:nrow)          :: &
  People_average   = 0.0
real , dimension(1:ncol)          :: &
  Subject_Average  = 0.0
integer :: r,c
  do r=1,nrow
    read 100,(exam_results(r,1:ncol)),people_average(r)
    100 format(1x,6(1x,f5.1),4x,f6.2)
  end do
  read *
  read 110, subject_average(1:ncol)
  110 format(1x,6(1x,f5.1))
  do r=1,nrow
    print
200,(exam_results(r,c),c=1,ncol),people_average(r)
    200 format(1x,6(1x,f5.1),' = ',f6.2)
  end do
  print *,'  ====  ====  ====  ====  ====  ===='
  print 210, subject_average(1:ncol)
  210 format(1x,6(1x,f5.1))
end program ch1005
```

Note also the use of

```
read *
```

to skip a line.

If you are on a UNIX or Linux system use diff to compare the input and output files. They should be the same.

10.9 Timing of Reading Formatted Files

A program to read a formatted file is shown below:

```
program ch1006
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x
```

```
real    , dimension(1:n) :: y
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
  open(unit=10,file='ch1006.txt',status='old')
  call cpu_time(t)
  t1=t
  comment=' Initial '
  print 100,comment,t1
  do i=1,n
    read(10,200) x(i)
    200 format(1x,i10)
  end do
  call cpu_time(t)
  t2=t-t1
  comment = ' I read '
  print 100,comment,t2
  do i=1,n
    read(10,300) y(i)
    300 format(1x,f10.0)
  end do
  call cpu_time(t)
  t3=t-t1-t2
  comment = ' r read '
  print 100,comment,t3
  100 format(1x,a,2x,f7.3)
  do i=1,10
    print *,x(i), ' ' , y(i)
  end do
end program ch1006
```

Some timing data from the Intel compiler follows:

```
  Initial         0.063
  I read          1.922
  r read          1.828
            1        1.000000
            2        2.000000
            3        3.000000
            4        4.000000
            5        5.000000
            6        6.000000
            7        7.000000
            8        8.000000
            9        9.000000
           10        10.00000
```

10.10 Timing of Reading Unformatted Files

The following is a program to read from an unformatted file:

```
program ch1007
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x
real    ,  dimension(1:n) :: y
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
  open(unit=10,file='ch1007.dat',form='unformatted')
  call cpu_time(t)
  t1=t
  comment=' Initial '
  print 100,comment,t1
  read(10) x
  call cpu_time(t)
  t2=t-t1
  comment = ' I read '
  print 100,comment,t2
  read (10) y
  call cpu_time(t)
  t3=t-t1-t2
  comment = ' r read '
  print 100,comment,t3
  100 format(1x,a,2x,f7.3)
  do i=1,10
    print *,x(i), ' ' , y(i)
  end do
end program ch1007
```

Some timing data from the Intel compiler follows.

```
  Initial        0.047
  I read        0.063
  r read        0.063
            1     1.000000
            2     2.000000
            3     3.000000
            4     4.000000
            5     5.000000
```

```
 6     6.000000
 7     7.000000
 8     8.000000
 9     9.000000
10    10.00000
```

10.11 Errors When Reading

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.

Chapter 18 looks at how we handle errors in input data, together with a more in-depth coverage of file I/O.

10.12 Flexible Input Using Internal Files

Sometimes external data does not have a regular structure and it is not possible to use the standard mechanisms we have covered so far in this chapter. Fortran provides something called internal files that allow us to solve this problem. The following example is based on a problem encountered whilst working at the following site

`http://www.shmu.sk/sk/?page=1`

They have data that is in the following format

```
#xxxxxxxxxx yyyyyyyyyy
```

where x and y can vary between 1 and 10 digits. The key here is to read the whole line (a maximum of 22 characters) and then scan the line for the blank character between the x and y digits.

We then use the `index` intrinsic to locate the position of the blank character. We now have enough information to be able to read the x and y integer data into the variables `n1` and `n2`.

```
program ch1008
implicit none
integer:: ib1,ib2
```

```
integer:: n1, n2
character(len=22):: buffer, buff1, buff2
! program to read a record of the form
! #xxxxxxxxxx yyyyyyyyyy
! so that integers n1 = xxxxxxxxxx n2 = yyyyyyyyyy
! where the number of digits varies from 1 to 10
! use internal files
   print *,"input micael's numbers"
   read(*,'(a)')buffer
   ib1 = index(buffer,' ')
   ib2 = len_trim(buffer)
    buff1 = buffer(2:ib1-1)
    buff2 = buffer(ib1+1:ib2)
    read(buff1,'(i10)')n1
    read(buff2,'(i10)')n2
    print*,'n1 = ',n1
    print*,'n2 = ',n2
end program ch1008
```

The statement

```
read(buff1,'(i10)')n1
```

reads from the string `buff1` and extracts the x number into the variable `n1`, and the statement

```
read(buff2,'(i10)')n2
```

reads from the string `buff2` and extracts the y number into the variable `n2`.

This is a very powerful feature and allows you to manage quite widely varying external data formats in files. `buff1` and `buff2` are called internal files in Fortran terminology.

10.13 Summary

Values may be read in from the keyboard, 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 blanks, although this can also be done through the BN and BZ formats.

Files can also be manipulated through rewind and backspace.

10.14 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 them 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, and 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? We will look at how we address this problem in Chapter 18.

Chapter 11
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.

11.1 Introduction

When you work interactively at a computer, you are working with files, files that contain programs, files that contain data, and perhaps files that are libraries. The file is fundamental to most modern 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, compilation listings, etc. They can be examined by printing them. They can also be transmitted around a computer system fairly easily. A file sent to a printer is a text file. Mail messages are generally plain text files. Note that when mail messages arrive in your mail box they will then typically contain additional nonprintable information.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_11, © Springer-Verlag London Limited 2012

2. Data files. These exist in two main forms: firstly those prepared by using an editor, (hence a text file) and those prepared using a package or program, in a computer readable form, but not directly readable by a human.
3. Binary, object or relocatable files, e.g., output from the compiler, satellite data. 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.

If you use a word processor then you will also have met files that are textual with additional nonprintable information.

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.

11.2 Data 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.txt')
```

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 be 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)
```

before we

```
open(unit=1,file='data2.txt')
```

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 ch1101
implicit none
real :: x
character (7) :: which
  open(unit=5,file='input')
  do
    write(unit=6,fmt='('' data set name, or end'')')
    read(unit=5,fmt='(a)') which
```

```
    if(which == 'end') exit
    open(unit=1,file=which)
    read(unit=1,fmt=100) x
!    ...
    close(unit=1)
  end do
end program ch1101
```

One useful feature of the open statement is that there are other parameters. What would happen, for example, if the file is not there? To take care of this you can use the iostat and status keywords, e.g.,

```
open(1,file='data.txt',iostat=filestat,status='old')
```

status can be equated to one of four values:

```
   status='old'
   status='new'
   status='scratch'
   status='unknown'
```

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, but if not create a 'new' one. if you omit the status=keyword altogether, the value 'unknown' will be assumed, if we use status='old' and the file is not present, this will cause an error which will be reflected in the value associated with the variable `open_file_status`. Consider the following example:

```
   ...
   open(unit=1,file='data.txt',iostat=filestat,status='o
ld')
   if (filestat > 0) then
      print *,' error opening file, please check'
      stop
   end if
   read(unit=1,fmt=100) x
   ...
```

The program will terminate after printing an appropriate error message. The standard defines that if an error occurs then IOSTAT will return a positive integer value. A value of zero is returned if there is no error.

11.3 Summary of Options on Open

unit: The unit number of the file to be opened.

iostat: The I/O status specifier designates a variable to store a value indicating the status of a data transfer operation. It takes the following form:

iostat=*i-var*

i-var – is a scalar integer variable. When a data transfer statement is executed, i-var is set to one of the following values:

- A positive integer indicating that an error condition occurred.
- A negative integer indicating that an end-of-file or end-of-record condition occurred. The actual values vary between compilers.
- Zero indicating no error, end-of-file, or end-of-record condition occurred.

Execution continues with the statement following the data transfer statement or the statement identified by a branch specifier (if any).

An end-of-file condition occurs only during execution of a sequential read statement; an end-of-record condition occurs only during execution of a non advancing read statement.

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 direct.

The two most common access mechanisms for files are sequential and direct. Consider a file with 1,000 records. to get at record 789 in a sequential file means reading or processing the first 788 records. to get at record 789 in a direct access file means using a record number to immediately locate record 789.

form: character expression specifying

formatted if the file is opened for formatted i/o

or

unformatted if the file is opened for unformatted 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.

As noted earlier data are maintained internally in a binary format, not immediately comprehensible by humans. When we wish to look at the data we must write it in a formatted fashion, i.e., as a sequence of printable ASCII characters – text, or the written word. This formatting will carry with it an overhead in terms of the time required to do it. It will also carry with it the penalty of conversion from one number base (internally binary) to another and also loss of significance due to rounding with whatever edit descriptors are used, e.g., writing out as F7.4.

If we are interested in reusing data on the same system and compiler then we can use the unformatted option and avoid both the time overhead (as there is no conversion between the internal and external formats) and the loss of significance associated with formatted data.

Please note that unformatted files are rarely portable between different computer systems, and sometimes even between different compilers on the same system.

We will look again at the use of unformatted files in Chapter ?? when we deal with efficiency and the space-time trade-off.

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.

11.4 More Foolproof I/O

Fortran provides a way of writing more foolproof programs involving I/O. This is done via the iostat keyword on the read statement. Consider the following:

```
program ch1102
implicit none
integer :: io_stat_number=-1
integer :: i
  do
   print*,'input integer i:'
   read (unit=*,fmt=10,&
          iostat=io_stat_number) i
   10 format(i3)
   print *,' iostat=',io_stat_number
   if (io_stat_number==0) exit
  end do
  print*,'i = ',i,' read successfully'
end program ch1102
```

The following data input should be tried and the values of IO_Stat_Number should be examined

- A valid three-digit number + [return] key
- A three-digit number with an embedded blank, e.g., 12 + [return] key
- [return] key only
- [CTRL] + Z
- Any other nonnumeric character on the keyboard
- 100200300 + [return] key
- [CTRL] + C

This will then enable you to write programs that handle common I/O errors.

Consider the following:

```
program ch1103
implicit none
integer , dimension(10) :: A =&
  (/-1,-1,-1,-1,-1,-1,-1,-1,-1,-1/)
integer :: io_stat_number=0
integer :: i
  open(unit=1,file='data.txt',status='old')
  do i=1,10
    read (unit=1,fmt=10,iostat=io_stat_number) A(i)
    10 format(I3)
    if (io_stat_number == 0) then
       cycle
    elseif (io_stat_number == -1) then
      print *,' end of file detected at line ',i
      print *,' please check data file'
      exit
    elseif (io_stat_number > 0 ) then
      print *,' non numeric data at line ',i
      print *,' please correct data file'
      exit
    endif
  end do
  do i=1,10
    print * , ' I = ',I,' A(i) = ',A(i)
  enddo
end program ch1103
```

The above program is system specific you will need to try it out with your compiler(s).

What happens with a completely blank line?

Note that in the above example the testing for the various conditions only exits the do loop for reading data from the file. This means that execution would continue with the statement immediately after the end do statement. This may not be what we want in all cases, and the exit may be replaced with a stop statement to terminate execution immediately.

11.5 Summary

The file is a fundamental entity within the operating system.

A file may be manipulated in Fortran by associating its 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.

11.6 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 eight columns allowed for each integer, with two spaces between integer fields.

   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 database 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. In 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 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 it 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?

Chapter 12
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 consider some of the reasons for the inclusion of functions in a programming language.
- To introduce, with examples, some of the predefined functions available in Fortran.
- To introduce a classification of intrinsic functions, generic, elemental, transformational.
- To introduce the concept of a user defined function.
- To introduce the concept of a recursive function.
- To introduce the concept of user defined elemental and pure functions.
- To briefly look at scope rules in Fortran for variables and functions.
- To look at internal user defined functions.

12.1 Introduction

The role of functions in a programming language and in the problem-solving process is considerable and includes:

- Allowing us to refer to an action using a meaningful name, e.g., sine(x) a very concrete use of abstraction.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_12, © Springer-Verlag London Limited 2012

- Providing a mechanism that allows us to break a problem down into parts, giving us the opportunity to structure our problem solution.
- Providing us with the ability to concentrate on one part of a problem at a time and ignore the others.
- Allowing us to avoid the replication of the same or very similar sections of code when solving the same or a similar subproblem which has the secondary effect of reducing the memory requirements of the final program.
- Allowing us to build up a library of functions or modules for solving particular subproblems, both saving considerable development time and increasing our effectiveness and productivity.

Some of the underlying attributes of functions are:

- They take parameters or arguments.
- The parameter can be an expression.
- A function will normally return a value and the value returned is normally dependent on the parameter(s).
- They can sometimes take arguments of a variety of types.

Most languages provide both a range of predefined functions and the facility to define our own. We will look at the predefined functions first.

12.2 An Introduction to Predefined Functions and Their Use

Fortran provides over a hundred intrinsic functions and subroutines. For the purposes of this chapter a subroutine can be regarded as a variation on a function. Subroutines are covered in more depth in a later chapter. They are used in a straightforward way. if we take the common trigonometric functions, sine, cosine and tangent, the appropriate values can be calculated quite simply by:

$$\mathrm{x} = \sin(\mathrm{y})$$
$$\mathrm{z} = \cos(\mathrm{y})$$
$$\mathrm{a} = \tan(\mathrm{y})$$

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.

12.2.1 Example 1: Simple Function Usage

A complete example is given below:

```
program ch1201
implicit none
real :: x
```

```
  print *,' type in an angle (in radians)'
  read *,x
  print *,' Sine of ', x ,' = ',sin(x)
end program ch1201
```

These functions are called intrinsic functions. A selection is follows:

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	k=mod(i,j)
	Remainder when I divided by j	
sqrt	Square root	x=sqrt(y)
exp	Exponentiation	y=exp(x)
log	Natural logarithm	x=log(y)
log 10	Common logarithm	x=log10(y)
sin	Sine	x=sin(y)
cos	Cosine	x=cos(y)
tan	Tangent	x=tan(y)
asin	Arc sine	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 C.

12.3 Generic Functions

All but four of the intrinsic functions and procedures are generic, i.e., they can be called with arguments of one of a number of kind types.

12.3.1 Example 2: The `abs` Generic Function

The following short program illustrates this with the `abs` intrinsic function:

```
program ch1202
implicit none
complex :: c=cmplx(1.0,1.0)
real    :: r=10.9
integer :: i=-27
  print *,abs(c)
  print *,abs(r)
  print *,abs(i)
end program ch1202
```

Type this program in and run it on the system you use.

It is now possible with Fortran for the arguments to the intrinsic functions to be arrays. It is convenient to categorise the functions into either elemental or transformational, depending on the action performed on the array elements.

12.4 Elemental Functions

These functions work with both scalar and array arguments, i.e., with arguments that are either single or multiple valued.

12.4.1 Example 3: Elemental Function Use

Taking the earlier example with the evaluation of sine as a basis, we have:

```
program ch1203
implicit none
real , dimension(5) :: x = (/1.0,2.0,3.0,4.0,5.0/)
  print *,' sine of ', x ,' = ',sin(x)
end program ch1203
```

In the above example the sine function of each element of the array x is calculated and printed.

12.5 Transformational Functions

Transformational functions are those whose arguments are arrays, and work on these arrays to transform them in some way.

12.5.1 Example 4: Simple Transformational Use

To highlight the difference between an element-by-element function and a transformational function consider the following examples:

```
program ch1204
implicit none
real , dimension(5) :: x = (/1.0,2.0,3.0,4.0,5.0/)
! elemental function
  print *,' Sine of ', x ,' = ',sin(x)
! Transformational function
  print *,' Sum of ', x ,' = ',sum(x)
end program ch1204
```

The sum function adds each element of the array and returns the sum as a scalar, i.e., the result is single valued and not an array.

12.5.2 Example 5: Intrinsic `dot_product` Use

The following program uses the transformational function `dot_product`:

```
program ch1205
implicit none
real , dimension(5) :: x = (/1.0,2.0,3.0,4.0,5.0/)
   print *,' Dot product of x with x is'
   print *,' ',dot_product(x,x)
end program ch1205
```

Try typing these examples in and running them to highlight the differences between elemental and transformational functions.

12.6 Notes on Function Usage

You should not use variables which have the same name as the intrinsic functions; e.g., what does sin(x) mean when you have declared sin to be a real array?

When a function has multiple arguments care must be taken to ensure that the arguments are in the correct position and of the appropriate kind 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)
```

12.7 Example 6: Easter

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 the 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 ch1206
implicit none
integer :: year, metcyc, century, error1, error2, day
integer :: epact, luna, temp
! 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
! calculating the year in the 19 year
! metonic cycle using variable metcyc
   metcyc = mod(year,19)+1
   if(year <= 1582)then
      day = (5*year)/4
      epact = mod(11*metcyc-4,30)+1
   else
!         calculating the century-century
      century = (year/100)+1
!         accounting for arithmetic inaccuracies
!         ignores leap years etc.
      error1 = (3*century/4)-12
      error2 = ((8*century+5)/25)-5
!         locating Sunday
      day = (5*year/4)-error1-10
!         locating the epact(full moon)
      temp = 11 * metcyc + 20 + error2 - error1
      epact = mod(temp,30)
      if(epact <= 0) then
         epact = 30 + epact
      endif
      if((epact == 25 .and. metcyc > 11) &
      .or. epact == 24)then
         epact = epact+1
      endif
   endif
!      finding the full moon
   luna= 44 - epact
   if (luna < 21) then
      luna = luna+30
   endif
!      locating easter Sunday
   luna = luna+7-(mod(day+luna,7))
```

```
!      locating the correct month
   if(luna > 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
   endif
end program ch1206
```

We have introduced a new statement here, the if then endif, and a variant the if then else endif. A more complete coverage is given in the chapter on control structures. The main point of interest is that the normal sequential flow from top to bottom can be varied. In the following case,

```
if (expression) then
   block of statements
endif
```

if the expression is true the block of statements between the if then and the endif is executed. If the expression is false then this block is skipped, and execution proceeds with the statements immediately after the endif.

In the following case,

```
if (expression) then
   block 1
else
   block 2
endif
```

if the expression is true block 1 is executed and block 2 is skipped. if the expression is false then block 2 is executed and block 1 is skipped. Execution then proceeds normally with the statement immediately after the endif.

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.

12.8 Intrinsic Procedures

An alphabetical list of all intrinsic functions and subroutines is given in Appendix C. This list provides the following information:

- Function name.
- Description.
- Argument name and type.

- Result type.
- Classification.
- Examples of use.

This appendix should be consulted for a more complete and thorough understanding of intrinsic procedures and their use in Fortran.

12.9 Supplying Your Own Functions

There are two stages here: firstly, to define the function and, secondly, to reference or use it. Consider the calculation of the greatest common divisor of two integers.

12.9.1 Example 7: Simple User Defined Function

The following defines a function to achieve this:

```
module gcd_module
contains
integer function gcd(a,b)
implicit none
integer , intent(in) :: a,b
integer :: temp
  if (a < b) then
    temp=a
  else
    temp=b
  endif
  do while ((mod(a,temp) /= 0) .or. (mod(b,temp) /=0))
    temp=temp-1
  end do
  gcd=temp
end function gcd
end module gcd_module
```

To use this function, you reference or call it with a form like:

```
program ch1207
use gcd_module
implicit none
integer :: i,j,result
integer :: gcd
  print *,' type in two integers'
  read *,i,j
  result=gcd(i,j)
  print *,' gcd is ',result
end program ch1207
```

We will start by talking about the actual function and then cover the following statements

```
module gcd_module
contains
..
end module gcd_module
```

and

```
use gcd_module
```

later.

The first line of the function

```
integer function gcd(a,b)
```

has a number of items of interest:

- Firstly the function has a type, and in this case the function is of type integer, i.e., it will return an integer value.
- The function has a name, in this case gcd.
- The function takes arguments or parameters, in this case `a` and `b`.

The structure of the rest of the function is the same as that of a program, i.e., we have declarations, followed by the executable part. This is because both a program and a function can be regarded as a program unit in Fortran terminology. We will look into this more fully in later chapters.

In the declaration we also have a new attribute for the integer declaration. The two parameters `a` and `b` are of type integer, and the `intent (in)` attribute means that these parameters will NOT be altered by the function. It is good programming practice for functions not to have side effects, i.e. not modify their arguments, and do no i/o.

The value calculated is returned through the function name somewhere in the body of the executable part of the function. In this case `gcd` appears on the left-hand side of an arithmetic assignment statement at the bottom of the function. The end of the function is signified in the same way as the end of a program:

```
end function gcd
```

We then have the program which actually uses the function `gcd`. In the program the function is called or invoked with `I` and `j` as arguments. The variables are called `a` and `b` in the function, and references to `a` and `b` in the function will use the values that `I` and `j` have respectively in the main program. We cover the area of argument association in the next section.

Note also a new control statement, the do while enddo. In the following case,

```
do while (expression)
  block of statements
enddo
```

the block of statements between the do while and the enddo is executed whilst the expression is true. There is a more complete coverage in Chap. 13.

We have two options here regarding compilation. Firstly, to make the function and the program into one file, and invoke the compiler once. Secondly, to make the function and program into separate files, and invoke the compiler twice, once for each file. With large programs comprising one program and several functions it is probably worthwhile to keep the component parts in different files and compile individually, whereas if it consists of a simple program and one function then keeping things together in one file makes sense.

12.10 An Introduction to the Scope of Variables, Local Variables and Interface Checking

One of the major strengths of Fortran is the ability to work on parts of a problem at a time. This is achieved by the use of program units (a main program, one or more functions and one or more subroutines) to solve discrete subproblems. Interaction between them is limited and can be isolated, for example, to the arguments of the function. Thus variables in the main program can have the same name as variables in the function and they are completely separate variables, even though they have the same name. Thus we have the concept of a local variable in a program unit.

In the example above `I, j, result,` are local to the main program. The declaration of `gcd` is to tell the compiler that it is an integer, and in this case it is an external function.

`a` and `b` in the function `gcd` do not exist in any real sense; rather they will be replaced by the actual variable values from the calling routine, in this case by whatever values `I` and `j` have. `temp` is local to `gcd`.

A common programming error in Fortran 66 and 77 was mismatches between actual and dummy arguments. Problems caused by this were often very subtle and hard to find.

Fortran 90 introduced a solution to the problem via the use of modules and contains statements. We have added

```
module gcd_module
contains
..
end module gcd_module
```

around the function definition, which contains the function in a module and the following statement in the main program

```
use gcd_module
```

provides an explicit interface (in Fortran terminology) that requires the compiler to check at compile time that the call is correct, i.e. that there are the correct number of parameters, they are of the correct type and in this case that the function return type is correct. We will cover this area in greater depth in later chapters.

12.11 Recursive Functions

There is an additional form of the function header that must be used when the function is recursive. Recursion means the breaking down of a problem into a simpler but identical subproblem. The concept is best explained with reference to an actual example. Consider the evaluation of a factorial, e.g., 5!. From simple mathematics we know that the following is true:

5!=5*4!
4!=4*3!
3!=3*2!
2!=2*1!
1!=1

and thus 5!=5*4*3*2*1 or 120.

12.11.1 Example 8: Recursive Factorial Evaluation

Let us look at a program with recursive function to solve the evaluation of factorials.

```
module factorial_module
implicit none
contains
recursive integer function factorial(i) result(answer)
implicit none
integer , intent(in):: I
   if (I==0) then
      answer=1
   else
      answer=i*factorial(I-1)
   end if
end function factorial
end module factorial_module

program ch1208
use factorial_module
implicit none
integer :: i, f
   print *,' type in the number, integer only'
   read *,I
   do while(i<0)
      print *,' factorial only defined for '
      print *,' positive integers: re-input'
      read *,I
```

```
   end do
   f=factorial(i)
   print *,' answer is', f
end program ch1208
```

What additional information is there? Firstly, we have an additional attribute on the function header that declares the function to be recursive. Secondly, we must return the result in a variable, in this case `answer`. Let us look now at what happens when we compile and run the whole program (both function and main program). If we type in the number 5 the following will happen:

- The function is first invoked with argument 5. The else block is then taken and the function is invoked again.
- The function now exists a second time with argument 4. The else block is then taken and the function is invoked again.
- The function now exists a third time with argument 3. The else block is then taken and the function is invoked again.
- The function now exists a fourth time with argument 2. The else block is then taken and the function is invoked again.
- The function now exists a fifth time with argument 1. The else block is then taken and the function is invoked again.
- The function now exists a sixth time with argument 0. The if block is executed and Answer = 1. This invocation ends and we return to the previous level, with Answer = 1*1.
- We return to the previous invocation and now answer = 2*1.
- We return to the previous invocation and now answer = 3*2.
- We return to the previous invocation and now answer = 4*6.
- We return to the previous invocation and now answer = 5*24.

The function now terminates and we return to the main program or calling routine. The answer 120 is the printed out.

Add a `print *,I` statement to the function after the last declaration and type the program in and run it. Try it out with 5 as the input value to verify the above statements.

Recursion is a very powerful tool in programming, and remarkably simple solutions to quite complex problems are possible using recursive techniques. We will look at recursion in much more depth in the later chapters on dynamic data types, and subroutines and modules.

12.12 Example 9: Recursive Version of gcd

The following is another example of the earlier gcd function but with the algorithm in the function replaced with an alternate recursive solution:

```
module gcd_module
implicit none
contains
recursive integer function gcd(i,j) result(answer)
implicit none
integer , intent(in) :: i,j
   if (j==0) then
      answer=I
   else
      answer=gcd(j,mod(i,j))
   endif
end function gcd
endmodule gcd_module

program ch1209
use gcd_module
implicit none
integer :: i,j,result
print *,' type in two integers'
read *,i,j
result=gcd(i,j)
print *,' gcd is ',result
end program ch1209
```

Try this program out on the system you work with, look at the timing information provided, and compare the timing with the previous example. The algorithm is a much more efficient algorithm than in the original example, and hence should be much faster. On one system there was a 20-fold decrease in execution time between the two versions.

Recursion is sometimes said to be inefficient, and the following example looks at a nonrecursive version of the second algorithm.

12.13 Example 10: After Removing Recursion

The following is a variant of the above, with the same algorithm, but with the recursion removed:

```
module gcd_module
implicit none
contains
integer function gcd(i,j)
implicit none
integer , intent(inout) :: i,j
integer :: temp
```

```
  do while (j/=0)
     temp=mod(i,j)
     i=j
     j=temp
  end do
  gcd=I
end function gcd
end module gcd_module

program ch1210
use gcd_module
implicit none
integer :: i,j,result
   print *,' type in two integers'
   read *,i,j
   result=gcd(i,j)
   print *,' gcd is ',result
end program ch1210
```

12.14 Internal Functions

An internal function is a more restricted and hidden form of the normal function definition.

Since the internal 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.

12.14.1 Example 11: Stirling's Approximation

In this example we use Stirling's approximation for large n,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

and a complete program to use this internal function is given below:

```
program ch1211
implicit none
real :: result,n,r
  print *,' type in n and r'
  read *,n,r
```

```
! number of possible combinations that can
! be formed when
! r objects are selected out of a group of n
! n!/r!(n-r)!
  result=stirling(n)/(stirling(r)*stirling(n-r))
  print *,result
  print *,n,r
contains
real function stirling (x)
  real , intent(in) :: x
  real , parameter :: pi=3.1415927, e =2.7182828
  stirling=sqrt(2.*pi*x) * (x/e)**x
end function stirling
end program ch1211
```

The difference between this example and the earlier ones lies in the contains statement. The function is now an integral part of the program and could not, for example, be used elsewhere in another function. This provides us with a very powerful way of information hiding and making the construction of larger programs more secure and bug free.

12.15 Pure Functions

We mentioned earlier that functions should not have side effects. If your functions do have side effects and are running the code on parallel systems we have the additional problem that it may not actually work! We would also like to be able to take advantage of automatic parallelisation if possible. In the following example we show how to do this using the `pure` prefix specification.

```
module gcd_module
implicit none
contains
    pure integer function gcd(a,b)
    implicit none
    integer , intent(in) :: a,b
    integer :: temp
      if (a < b) then
        temp=a
      else
        temp=b
      endif
      do while ((mod(a,temp) /= 0) &
         .or. (mod(b,temp) /=0))
          temp=temp-1
      end do
```

```
      gcd=temp
  end function gcd
end module gcd_module

program ch1212
use gcd_module
implicit none
integer :: i,j,result
  print *,' type in two integers'
  read *,i,j
  result=gcd(i,j)
  print *,' gcd is ',result
end program ch1212
```

Subroutines can also be made pure.

12.15.1 Pure Constraints

The following are some of the constraints on pure procedures

- a dummy argument must be intent(in)
- local variables may not have the save attribute
- no i/o must be done in the procedure
- any procedures referenced must be pure
- you cannot have a stop statement in a pure procedure

The above information should be enough to write simple pure functions.

12.16 Elemental Functions

Fortran 77 introduced the concept of generic intrinsic functions. Fortran 90 added elemental intrinsic functions and the ability to write generic user defined functions. Fortran 95 squared the circle and enabled us to write elemental user defined functions. Here is an example to illustrate this.

```
module reciprocal_module
contains
real elemental function reciprocal(a)
implicit none
real , intent(in) :: a
  reciprocal=1.0/a
end function reciprocal
end module reciprocal_module
```

```
program ch1213
use reciprocal_module
implicit none
real :: x=10.0
real , dimension(5) :: y=[1.0,2.0,3.0,4.0,5.0]
  print *,' reciprocal of x is ',reciprocal(x)
  print *,' reciprocal of y is ',reciprocal(y)
end program ch1213
```

Here is the output from one compiler.

```
   reciprocal of x is    0.1000000
   reciprocal of y is    0.9999999       0.5000000
  0.3333333
    0.2500000       0.2000000
```

Hence we can call our own elemental functions with both scalar and array arguments.

Elemental functions require the use of explicit interfaces, and we have therefore used modules to achieve this.

12.17 Resumé

There are a large number of Fortran supplied functions and subroutines (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 arguments. Others are restricted to a particular type of argument. Appendix C should be consulted for a fuller coverage concerning the rules that govern the use of the intrinsic functions and procedures.

When the intrinsic functions are inadequate, it is possible to write user defined functions. Besides expanding the scope of computation, such functions aid in problem visualisation and logical subdivision, may reduce duplication, and generally help in avoiding programming errors.

In addition to separately defined user functions, internal functions may be employed. These are functions 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 nonrecursive functions 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.

We have also covered briefly the concept of scope for a variable, local variables, and argument association. This area warrants a much fuller coverage and we will do this after we have covered subroutines and modules.

12.18 Formal Syntax

The syntax of a function is:

```
[function prefix] function_statement &
[result (result_name) ]
[specification part]
[execution_part]
[internal sub program part]
end [function [function name]]
```

and prefix is:

```
[type specification] recursive
```

or

```
[recursive] type specification
```

and the function_statement is:

```
function function_name ([dummy argument name list])
```

[ ] represent optional parts to the specification.

The simple syntax for a module as we have used them in this chapter is

```
module module_name
contains
...
end module module_name
```

and

```
use module_name
```

in the calling routine.

12.19 Rules and Restrictions

The type of the function must only be specified once, either in the function statement or in a type declaration.

The names must match between the function header and end function function name statement.

If there is a result clause, that name must be used as the result variable, so all references to the function name are recursive calls.

The function name must be used to return a result when there is no result clause. We will look at additional rules and restrictions in later chapters.

12.20 Problems

1. 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`!
2. Create a table which gives the sines, cosines and tangents for –1 to 91 degrees in 1 degree intervals. Remember that the arguments have to be in radians. What value will you give π? One possibility is π = 4*atan(1.0). Pay particular attention to the following angle ranges:

   - −1,0,+1
   - 29,30,31
   - 44,45,46
   - 59,60,61
   - 89,90,91

   What do you notice about sine and cosine at 0 and 90 degrees? What do you notice about the tangent of 90 degrees? Why do you think this is?
   Use a calculator to evaluate the sine, cosine at 0 and 90 degrees. do the same for the tangent at 90 degrees. Does this surprise you?
   Repeat using a spreadsheet, e.g., Excel.
   Are you surprised?
   Repeat the Fortran program using one or more real kind types.
3. Write a program that will read in the lengths a and b of a right-angled triangle and calculate the hypotenuse c. Use the Fortran **sqrt** intrinsic.
4. Write a program that will read in the lengths a and b of two sides of a triangle and the angle between them θ (in degrees). Calculate the length of the third side c using the cosine rule:

$$c^2 = a^2 + b^2 - 2abcos(\theta)$$

5. Write a function to convert an integer to a binary character representation. It should take an integer argument and return a character string that is a sequence of zeros and ones. Use the program in Chap. 5 as a basis for the solution.

12.21 Bibliography

Abramowitz, M., Stegun, I.: Handbook of Mathematical Functions. Dover, New York (1968)

This book contains a fairly comprehensive collection of numerical algorithms for many mathematical functions of varying degrees of obscurity. It is a widely used source.

Association of Computing Machinery (ACM)
Collected Algorithms, 1960–1974
Transactions on Mathematical Software, 1975 – A good source of more specialised algorithms. Early algorithms tended to be in Algol, Fortran now predominates.

12.21.1 Recursion and Problem Solving

The following are a number of books that look at the role of recursion in problem solving and algorithms.

Hofstadter, D.R.: Gödel, Escher, Bach—An Eternal Golden Braid. Harvester Press, London (1979)
The book provides a stimulating coverage of the problems of paradox and contradiction in art, music and mathematics using the works of Escher, Bach and Gödel, and hence the title. There is a whole chapter on recur sive structures and processes. The book also covers the work of Church and Turing, both of whom have made significant contributions to the theory of computing.
Kruse, R.L.: Data Structures and Program Design. Prentice-Hall, Englewood Cliffs (1994)
Quite a gentle introduction to the use of recursion and its role in problem solving. Good choice of case studies with explanations of solutions. Pascal is used.
Sedgewick, R.: Algorithms in Modula 3. Addison-Wesley, Reading (1993)
Good source of algorithms. Well written. The gcd algorithm was taken from this source.
Vowels, R.A.: Algorithms and Data Structures in F and Fortran. Unicomp, Tucson (1998)
The only book currently that uses Fortran 90/95 and F. Visit the Fortran web site for more details. They are the publishers.
http://www.fortran.com/fortran/market.html

Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs (1976)
In the context of this chapter the section on recursive algorithms is a very worthwhile investment in time.
Wood, D.: Paradigms and Programming in Pascal. Computer Science Press, Rockville (1984)
contains a number of examples of the use of recursion in problem solving. Also provides a number of useful case studies in problem solving.

Chapter 13
Control Structures

Summarizing: as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure.

Edsger W. Dijkstra, Structured Programming

Aims

The aims of this chapter are to introduce:

- Selection among various courses of action as part of the algorithm.
- The concepts and statements in Fortran needed to support the above:

  Logical expressions and logical operators.
  One or more blocks of statements.

- The if then endif construct.
- The if then else if endif construct.
- To introduce the case statement with examples.
- To introduce the do loop, in three forms with examples, in particular:

  The iterative do loop.
  The do while form.
  The do … if then exit end do or repeat until form.
  The cycle statement.
  The exit statement.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_13, © Springer-Verlag London Limited 2012

13.1 Introduction

When we look at this area it is useful to gain some historical perspective concerning the control structures that are available in a programming language.

At the time of the development of Fortran in the 1950s there was little theoretical work around and the control structures provided were very primitive and closely related to the capability of the hardware.

At the time of the first standard in 1966 there was still little published work regarding structured programming and control structures. The seminal work by Dahl, Dijkstra and Hoare was not published until 1972.

By the time of the second standard there was a major controversy regarding languages with poor control structures like Fortran which essentially were limited to the goto statement. The facilities in the language had led to the development and continued existence of major code suites that were unintelligible, and the pejorative term spaghetti was applied to these programs. Developing an understanding of what a program did became an almost impossible task in many cases.

Fortran missed out in 1977 on incorporating some of the more modern and intelligible control structures that had emerged as being of major use in making code easier to understand and modify.

It was not until the 1990 standard that a reasonable set of control structures had emerged and became an accepted part of the language. The more inquisitive reader is urged to read at least the work by Dahl, Dijkstra and Hoare to develop some understanding of the importance of control structures and the role of structured programming. The paper by Knuth is also highly recommended as it provides a very balanced coverage of the controversy of earlier times over the goto statement.

13.2 Selection Among Courses of Action

In most problems you need to choose among 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 below.

13.2.1 The Block if Statement

The following short example illustrates the main ideas:

```
. wake up
.
. check the date and time
if (Today = = 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 >= 10.1)
```

Test if Alpha is greater than or equal to 10.1

```
(Balance <= 0.0)
```

Test if overdrawn

```
(( Today == Saturday).OR.( Today == Sunday))
```

Test if today is Saturday or Sunday

```
((Actual - Calculated)<= 1.0E-6)
```

Test if Actual minus Calculated is less than or equal to 1.0E-6

Fortran has the following relational and logical operators:

operator	Meaning	type
= =	Equal	Relational
/=	Not equal	Relational
>=	Greater than or equal	Relational
<=	Less than or equal	Relational
<	Less than	Relational
>	Greater than	Relational
.AND.	And	Logical
.OR.	Or	Logical
.NOT.	Not	Logical

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.

Use of logical expressions and logical variables (something not mentioned so far) is covered again in a later chapter on logical 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 (Balance > 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 an elseif statement. Consider the following example:

```
if (today == Monday) then
    .
elseif (today == Tuesday) then
    .
elseif (today == Wednesday) then
    .
elseif (today == Thursday) then
    .
elseif (today == Friday) then
    .
elseif (today == Saturday) then
    .
elseif (today == 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 < 2)then
    ...
  elseif(I < 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 are in certain well-specified ranges. The program then fails when the data go outside this range. It makes no sense to have a day other than Monday, Tuesday, Wednesday, Thursday, Friday, Saturday or Sunday.

13.2.1.1 Example 1: Quadratic Roots

A quadratic equation is:

$$ax^2 + bx + c = 0$$

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 or 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 ch1301
implicit none
real :: A , B , C , Term , A2 , Root1 , root2
!
!   a b and c are the coefficients of the terms
!   a*x**2+b*x+c
!   find the roots of the quadratic, root1 and root2
!
  print*,' give the coefficients a, b and c'
  read*,a,b,c
  term = b*b - 4.*a*c
  a2 = a*2.
```

```
! if term < 0, roots are complex
! if term = 0, roots are equal
! if term > 0, roots are real and different
  if(term < 0.0)then
    print*,' roots are complex'
  elseif(term > 0.0)then
    term = sqrt(term)
    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 program ch1301
```

13.2.1.2 Note

Given the understanding you now have about real arithmetic and finite precision will the else block above ever be executed?

13.2.1.3 Example 2: Date Calculation

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 ch1302
implicit none
integer :: Year , N , Month , Day , T
!
! calculates day and month from year and
! day-within-year
! t is an offset to account for leap years.
! Note that the first criteria is division by 4
! but that centuries are only
! leap years if divisible by 400
! not 100 (4 * 25) alone.
!
  print*,' year, followed by day within year'
  read*,Year,N
!   checking for leap years
```

```
  if ((Year/4)*4 == Year) then
    T=1
    if ((Year/400)*400 == Year) then
      T=1
    ELSEIF ((Year/100)*100 == Year) then
      T=0
    endif
  else
      T=0
  endif
!    accounting for February
  if(N > (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 program ch1302
```

13.2.2 The Case Statement

The case statement provides a very clear and expressive selection mechanism between two or more courses of action. Strictly speaking it could be constructed from the if then elseif endif statement, but with considerable loss of clarity. Remember that programs have to be read and understood by both humans and compilers!

13.2.2.1 Example 3: Simple Calculator

```
Program ch1303
implicit none
!
! Simple case statement example
!
integer :: I,J,K
character :: operator
  do
    print *,' type in two integers'
    read *, I,J
    print *,' type in operator'
    read '(A)',operator
    Calculator : &
```

```
    select case (operator)
      case ('+') Calculator
        K=I+J
        print *,' Sum of numbers is ',K
      case ('-') Calculator
        K=I-J
        print *,' Difference is ',K
      case ('/') Calculator
        K=I/J
        print *,' Division is ',K
      case ('*') Calculator
      K=I*J
      print *,' Multiplication is ',K
    case default Calculator
      exit
    end select Calculator
  end do
end program ch1303
```

The user is prompted to type in two integers and the operation that they would like carried out on those two integers. The case statement then ensures that the appropriate arithmetic operation is carried out. The program terminates when the user types in any character other than +, –, * or /.

The case default option introduces the exit statement. This statement is used in conjunction with the do statement. When this statement is executed control passes to the statement immediately after the matching end do statement. In the example above the program terminates, as there are no executable statements after the end do.

13.2.2.2 Example 4: Counting Vowels, Consonants, etc

This example is more complex, but again is quite easy to understand. The user types in a line of text and the program produces a summary of the frequency of the characters typed in:

```
program ch1304
implicit none
!
! Simple counting of vowels, consonants,
! digits, blanks and the rest
!
integer :: Vowels=0 , Consonants=0, Digits=0
integer :: Blank=0, Other=0, I
character :: Letter
```

```
character (LEN=80) :: Line
  read '(A)', Line
  do i=1,80
    Letter=Line(I:I)
! the above extracts one character at position I
    select case (Letter)
      case ('A','E','I','O','U', &
            'a','e','i','o','u')
        Vowels=Vowels + 1
      case ('B','C','D','F','G','H', &
             'J','K','L','M','N','P', &
             'Q','R','S','T','V','W', &
             'X','Y','Z',             &
             'b','c','d','f','g','h', &
             'j','k','l','m','n','p', &
             'q','r','s','t','v','w', &
             'x','y','z')
        Consonants=Consonants + 1
      case ('1','2','3','4','5','6','7','8','9','0')
        Digits=Digits + 1
      case (' ')
        Blank=Blank + 1
      case default
        Other=Other+1
    end select
  end do
  print *, ' Vowels = ', Vowels
  print *, ' Consonants = ', Consonants
  print *, ' Digits = ', Digits
  print *, ' Blanks = ',Blank
  print *, ' Other characters = ', Other
end program ch1304
```

13.3 The Three Forms of the do Statement

You have already been introduced in the chapters on arrays to the iterative form of the do loop, i.e.,

```
do Variable = Start, End, Increment
        block of statements
end do
```

A complete coverage of this form is given in the three chapters on arrays. There are two additional forms of the block do that complete our requirements:

```
        do while (Logical Expression)
                block of statements
        enddo
```

and

```
        do
                block of statements
                if (Logical Expression) exit
        end do
```

The first form is often called a while loop as the block of statements executes whilst the logical expression is true, and the second form is often called a repeat until loop as the block of statements executes until the statement is true.

Note that the while block of statements may never be executed, and the repeat until block will always be executed at least once.

13.3.1 Example 5: Sentinel Usage

The following example shows a complete program using this construct:

```
program ch1305
implicit none
! this program picks up the first occurrence
! of a number in a list.
! a sentinel is used, and the array is 1 more
! than the max size of the list.
integer , allocatable , dimension(:) :: A
integer :: Mark
integer :: i,Howmany
  open (unit=1,file='data.txt')
  print *,' What number are you looking for?'
  read  *, Mark
  print *,' How many numbers to search?'
  read  *,Howmany
  allocate(A(1:Howmany+1))
  read(unit=1,fmt=*) (A(i),I=1,Howmany)
  I=1
  A(Howmany+1)= Mark
  do while(Mark /= A(i))
    I=I+1
  end do
```

```
  if(I == (Howmany+1)) then
    print*,' item not in list'
  else
    print*,' item is at position ',I
  endif
end program ch1305
```

The repeat until construct is written in Fortran as:

```
  do
  ...
  ...
    if (Logical Expression) exit
  end do
```

There are problems in most disciplines that require a numerical solution. The two main reasons for this are either that 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. A program extract to illustrate this follows:

```
   real , parameter :: tol=1.0e-6
   .
   do
      ...
      change=
      ...
      if (change <= tol) exit
   end do
```

Here the value of the tolerance is set to 1.0E–6. Note again the use of the exit statement. The do end do block is terminated and control passes to the statement immediately after the matching end do.

13.3.2 Cycle and Exit

These two statements are used in conjunction with the block do statement. You have seen examples above of the use of the exit statement to terminate the block do, and pass control to the statement immediately after the corresponding end do statement.

The cycle statement can appear anywhere in a block do and will immediately pass control to the start of the block do. Examples of cycle and exit are given in later chapters.

13.3.3 Example 6: e**x Evaluation

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

$$1+\sum_{n=1}^{\infty}\frac{x^{n-1}}{(n-1)!}\frac{x}{n}$$

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:

```
module etox_module
implicit none
contains
real function etox(x)
implicit none
real :: term
real , intent(in) :: x
integer :: nterm
real , parameter ::tol = 1.0E-6
  etox=1.0
  term=1.0
  nterm=0
  do
    nterm = nterm +1
    term = ( x / nterm) * term
    etox = etox + Term
    if (abs(term) <= tol)exit
  end do
end function etox
end module etox_module

program ch1306
use etox_module
implicit none
real , parameter :: x=1.0
real :: y
```

```
  print *,' Fortran intrinsic ',exp(x)
  y=etox(x)
  print *,' User defined etox ',y
end program ch1306
```

The whole program compares the user defined function with the Fortran intrinsic exp function.

13.3.4 Example 7: Wave Breaking on an Offshore Reef

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.

Both types of loops are combined in this 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 signs. The while loop ensures that the evaluations are of different signs, 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

Here is the program:

```
program ch1307
implicit none
real :: Hi , Hr , Hlow , High , Half , Xl
real :: Xh , Xm , D
real , parameter :: Tol=1.0E-6
! problem - find hi from expression given
! in function f
! F=A*(1.0-0.8*EXP(-0.6*C/A))-B
! hi is incident wave height        (c)
! hr is reformed wave height        (b)
! d is water depth at terrace edge  (a)
   print*,' Give reformed wave height, and water depth'
   read*,Hr,d
!
! for Hlow- let Hlow=hr
! for high- let high=Hlow*2.0
!
```

```
! check that signs of function results are different
!
  Hlow = hr
  high = hlow*2.0
  xl = f( hlow, hr, d)
  xh = f( high, hr, d)
!
  do while ((xl*xh)  >= 0.0)
     high = high*2.0
     xh = f(high,hr,d)
  end do
!
  do
    half=(hlow+high)*0.5
    xm=f(half,hr,d)
    if((xl*xm) < 0.0)then
       xh=xm
       high=half
    else
       xl=xm
       hlow=half
    endif
    if(abs(high-hlow)<= tol)exit
  end do
  print*,' Incident Wave Height Lies Between'
  print*,Hlow,' and ',high,' metres'
contains
real function f(a,b,c)
implicit none
real , intent (in) :: a
real , intent (in) :: b
real , intent (in) :: c
   f=a*(1.0-0.8*exp(-0.6*c/a))-b
end function f
end program ch1307
```

13.4 Summary

You have been introduced in this chapter to several control structures and these include:

- The block if.
- The if then else if.

- The case construct.
- The block do in three forms:
  - The iterative do or do variable = start,end,increment … end do.
  - The while construct, or do while … end do.
  - The repeat until construct, or do … if then exit end do.
- The cycle and exit statements, which can be used with do statement in all three forms:
  - The do variable = start,end,increment … end do.
  - The while construct, or do while … end do.
  - The repeat until construct, or do … if then exit end do.

These constructs are sufficient for solving a wide class of problems. There are other control statements available in Fortran, especially those inherited from Fortran 66 and Fortran 77, but those covered here are the ones preferred. We will look in Chap. 35 at one more control statement, the so-called goto statement, with recommendations as to where its use is appropriate.

13.4.1 Control Structure Formal Syntax

case

```
select case ( case variable )
   [ case case selector
      [executable construct ] ... ] ...
   [ case DEFAULT
   [ executable construct ]
end select
```

do

```
do [ label ]
   [executable construct ] ...
do termination

do [ label ] [ , ] loop variable = initial value ,
final value , [ increment ]
   [executable construct ] ...
do termination

do [ label ] [ , ] while (scalar logical expression )
   [executable construct ] ...
do termination
```

if

```
if ( scalar logical expression ) then
   [executable construct ] ...
[ else if ( scalar logical expression then
   [executable construct ] ... ] ...]
[ else
     [executable construct ] ...]
end if
```

13.5 Problems

1. Rewrite the program for the period of a pendulum. The new program should print out the length of the pendulum and period, for pendulum lengths from 0 to 100 cm in steps of 0.5 cm. The program should incorporate a function for the evaluation of the period.
2. Write a program to read an integer that must be positive. Hint. Use a do while to make the user re-enter the value.
3. Using functions, do the following:
   - Evaluate n! from n = 0 to n = 10.
   - Calculate 76!
   - Now calculate (x**n)/n!, with x = 13.2 and n = 20.
   - Now do it another way.
4. The program ch1307 is taken from a real example. In the particular problem, the reformed wave height was 1 m, and the water depth at the reef edge was 2 m. 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% 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.
5. What is the effect of using int on negative real numbers? Write a program to demonstrate this.
6. 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.
7. 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.
   The physical world has many examples in which processes require that some threshold be overcome before they begin operation: critical mass in nuclear reac-

tions, 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 tries to restrict the range of calculation, whilst illustrating the possibilities of decision making.

8. If a cubic equation is expressed as

$$ax^3 + bx^2 + cx + d = 0$$

and we let

$$\Delta = 18abcd - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2$$

We can determine the nature of the roots as follows

$\Delta > 0$: three distinct real roots

$\Delta = 0$: has a multiple root and all roots are real

$\Delta < 0$: 1 real root and 2 non real complex conjugate roots

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

9. 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 s^{-2}). If B is less than 0.003, 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 is swell waves, with wave heights of 1–2 m and wave periods of 10–15 s. During storms, the wave period is generally shorter, say 6–8 s, and the wave heights higher, 3–5 m. 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 ° (m = 0.018), but towards the high-tide

mark, the slope increases dramatically, to say 10° or more (m = 0.18). What changes in wave type will be observed as the tide comes in?

13.6 Bibliography

Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, London (1972)

- This is the original text, and a must. The quote at the start of the chapter by Dijkstra summarises beautifully our limitations when programming and the discipline we must have to master programming successfully.

Knuth, D.E.: Structured Programming with goto Statements, in Current Trends in Programming Methodology, vol 1. Prentice-Hall (1977)

- The chapter by Knuth provides a very succinct coverage of the arguments for the adoption of structured programming, and dispels many of the myths concerning the use of the goto statement. Highly recommended.

Chapter 14
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.

14.1 Introduction

For each type in a programming language there are the following concepts:

- Values are drawn from a finite domain.
- There are a restricted number of operations defined for each type.

For the numeric types we have already met, integers and reals:

- The values are either drawn from the domain of integer numbers or the domain of real numbers.
- The valid operations are addition, subtraction, multiplication, division and exponentiation.

For the character data type the basic unit is an individual character. The complete Fortran character set is given in Sect. 4.6 in Chap. 4. This provides us with 95 printing characters. Other characters may be available. The Wikipedia entry

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_14, © Springer-Verlag London Limited 2012

```
http://en.wikipedia.org/wiki/Character_encoding
```

has quite detailed information on how complex this area actually is.

As the most common current internal representation for the character data type uses 8 bits this should provide access to 256 (2^8) characters. However, there is little agreement over the encoding of these 256 possible characters, and the best you can normally assume is access to the ASCII character set, which is given in Appendix B. One of the problems at the end of this chapter looks at determining what characters one has available.

The only operations defined are concatenation (joining character strings together) and comparison.

We will look into the area of character sets in more depth later in this chapter. We can declare our character variables:

```
character :: a, string, line
```

Note that there is no default typing of the character variable (unlike integer and real data types), 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 one item of information:

```
character (10) :: A
character (16) :: string
character (80) :: line
```

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 can 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 (') or quotation mark (") is needed to indicate that this is a character string (otherwise the assignments would have looked like invalid variable names).

14.2 Character Input

In an earlier chapter we saw how we could use the read * and print * statements to do both numeric and character input and output or I/O. When we use this form of the statement we have to include any characters we type within delimiters (either the apostrophe ' or the quotation mark "). This is a little restricting and there is a slightly more complex form of the read statement that allows one to just type the string on its own. The following two programs illustrate the differences:

```
program ch1401
!
! Simple character i/o
!
character (80) :: line
  read *, line
  print *, line
end program ch1401
```

This form requires enclosing the string with delimiters. Consider the next form:

```
program ch1402
!
! Simple character i/o
!
character (80) :: line
  read '(a)' , line
  print *,line
end program ch1402
```

With this form one can just type the string in and input terminates with the carriage return key. The additional syntax involves '(A)' where '(A)' is a character edit descriptor. The simple examples we have used so far have used implied format specifiers and edit descriptors. For each data type we have one or more edit descriptors to choose from. For the character data type only the A edit descriptor is available.

14.3 Character Operators

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 (5) :: first, second
character (10) :: third
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' i.e., 'blin', and third will therefore contain 'three blin'.

What would happen if we assigned a character variable of length ‘n’ a string which was shorter than n? For example,

```
character (4) :: c2
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 (8) :: jj
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 – substrings. The actual notation for doing this is a little strange at first, but it is very powerful. To extract a substring we must reference two items:

- The position in the string at which the substring begins,

and

- The position at which it ends.

e.g.,

```
string='share and enjoy'
```

14.4 Character Substrings

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 substring 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 to be of character type, of some appropriate length.

character variables may also form arrays:

```
character (10) , dimension(20) :: A
```

sets up a character array of 20 elements, where each element contains 10 characters. In order to extract substrings from these array elements, we need to know where the array reference and the substring reference are placed. The array reference comes first, so that

```
do i=1,20
   first=a(i)(1:1)
endo
```

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 blanks out of a string:

```
program ch1403
implicit none
  character(80) :: String, Strip
  integer :: ipos,i,length=80
  ipos=0
  print *,' type in a string'
  read '(a)',string
```

```
  do i=1,length
    if(string(i:i) /= ' ') then
      ipos=ipos+1
      strip(ipos:ipos)=string(i:i)
    endif
  end do
  print*,string
  print*,strip
end program ch1403
```

14.5 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:

```
program ch1404
implicit none
character (80) :: Line
integer :: i
  do
    read '(A)', Line
    I=Index(Line,'Geology')
    if (I /= 0) then
      print *, ' String Geology found at position ',I
      print *, ' in line ', Line
      exit
    endif
  enddo
end program ch1404
```

There are two things to note about this program. Firstly the index function 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 them in some way. Secondly, if the string does not occur, the result of the index function is zero, and given the infinite loop (do end do) the program will crash at run time with an end of file error message. This isn't good programming practice.

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 following example illustrates the use of both len and len_trim:

```
program ch1405
implicit none
character (len=20) :: name
integer :: name_length
  print *,' type in your name'
  read '(a)',name
!
! show len first
!
  name_length=len(name)
  print *,' name length is ',name_length
  print *,' ',name(1:name_length),'<-end is here'
  name_length=len_trim(name)
  print *,' name length is ',name_length
  print *,' ',name(1:name_length),'<-end is here'
end program ch1405
```

14.6 Collating Sequence

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 63 defined characters, the machine you use will generally have more; 95 printing characters is a typical minimum number. On this type of machine the weights would vary from 0 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, 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 0–9), as do A to Z and 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  I=65,90
   print*,char(i)
enddo
```

This idea of a weighting can 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') > ichar('b')) then
```

but these functions can take character string arguments of any length. They are not restricted to single characters.

These functions provide very powerful tools for the manipulation of characters, and open up wide areas of nonnumerical computing through Fortran. Text formatting and word processing applications may now be tackled (conveniently ignoring the fact that lower-case characters may not be available).

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 by Kernighan and Plauger contains many interesting uses of characters in Fortran.

14.7 Finding Out About the Character Set Available

The following program prints out the characters between 32 and 127.

```
program ch1406
implicit none
integer :: i
  do i=32,62
```

```
    print *,i,char(i),i+32,char(i+32),i+64,char(I+64)
  end do
  I=63
  print *,i,char(i),i+32,char(i+32),i+64,'del'
  I=64
  print *,i,char(i),i+32,char(I+32)
end program ch1406
```

This is the output from the Intel compiler under Windows.

```
32             64 @          96 `
33 !           65 A          97 a
34 "           66 B          98 b
35 #           67 C          99 c
36 $           68 D         100 d
37 %           69 E         101 e
38 &           70 F         102 f
39 '           71 G         103 g
40 (           72 H         104 h
41 )           73 I         105 I
42 *           74 J         106 j
43 +           75 K         107 k
44 ,           76 L         108 l
45 -           77 M         109 m
46 .           78 N         110 n
47 /           79 O         111 o
48 0           80 P         112 p
49 1           81 Q         113 q
50 2           82 R         114 r
51 3           83 S         115 s
52 4           84 T         116 t
53 5           85 U         117 u
54 6           86 V         118 v
55 7           87 W         119 w
56 8           88 X         120 x
57 9           89 Y         121 y
58 :           90 Z         122 z
59 ;           91 [         123 {
60 <           92 \         124 |
61 =           93 ]         125 }
62 >           94 ^         126 ~
63 ?           95 _         127 del
64 @           96 `
```

Try this program out on the system you use. Do the character sets match?

14.8 Scan Function Example

The following program uses the scan function to locate the position of all of the blanks in a string. The syntax of the simple form we use in the program is given below.

- scan(string,set) – Scans a string for any one of the characters in a set of characters.

```
program ch1407
implicit none
character (1024) :: string01
character (1)     :: set=' '
integer :: i
integer :: l
integer :: start,end
  string01 = &
  "The important issue about a language, is not so"
  string01  = trim(string01)  // " " // &
  "much what features the language possesses, but"
  string01  = trim(string01)  // " " // &
  "the features it does possess, are sufficient, to"
  string01  = trim(string01)  // " " // &
  "support the desired programming styles, in the"
  string01  = trim(string01)  // " " // &
  "desired application areas. "
  l = len(trim(string01))
  print *,' Length of string is = ',l
  print *,' String is'
  print *,trim(string01)
  start=1
  end=l
  print *,' Blanks at positions '
  do
    i=scan(string01(start:end),set)
    start=start+i
    if (i==0) exit
    write(*,10,advance='no')start-1
    10 format(i5)
  end do
end program ch1407
```

Note the use of the trim function when using the concatenation operator to initialise the string to the text we want.

The output from one compiler is given below.

```
   Length of string is=      217
   String is
  The important issue about a language, is not so much
what features the language possesses, but the
features it does possess, are sufficient, to support
the desired programming styles, in the desired
application areas.
  Blanks at positions
    4    14    20    26    28    38    41   45   48     53
58   67    71   80    91    95
   99   108  111   116  125   129  141  144  152  156
164  176  184  187  191  199
  211
```

The text in this program is used in two problems at the end of this chapter.

14.9 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 (') or quotation marks ("). Within a character string, the blank is a significant character.

Character strings may be joined together (concatenated) with the // operator.

Substrings occurring within character strings may be also be manipulated. There are a number of functions especially for use with characters:

- achar – return the character in the ASCII character set
- adjustl – adjust left, remove leading blanks, add trailing blanks
- adjustr – adjust right – remove trailing blanks, insert leading blanks
- char – return the character in the processor collating sequence
- iachar – as above but in the ASCII character set
- index – locate one string in another
- len – character length including trailing blanks
- len_trim – character length without the trailing blanks
- lle – lexically less than or equal to
- lge – lexically greater than or equal to
- lgt – lexically greater than
- llt – lexically less than
- repeat – concatenate several copies of a string
- scan – scans a string for anyone of the characters in the set
- trim – remove the trailing blanks
- verify – verify that a set of characters contains all the characters in a string

A detailed explanation is given in Appendix E.

14.10 Problems

1. Suggest some circumstances where PRIME="" might be useful. What other alternative is there and why do you think we use that instead?
2. Write a program to write out the weights for the Fortran character set. Modify this program to print out the weights of the complete implementation defined character set for your version of Fortran. Is it ASCII? If not, how does it differ?
3. Write a program that produces the following output.

```
!
"#
$%&
'()*
+,-./
012345
6789:;<
=>?@ABCD
EFGHIJKLM
NOPQRSTUVW
XYZ[\]^_`ab
cdefghijklmn
opqrstuvwxyz{
|}~
```

   We assume the ASCII character set in this example.
4. Modify the above program to produce the following output.

```
          !
         "#$
        %&'()
       *+,-./0
      123456789
     :;<=>?@ABCD
    EFGHIJKLMNOPQ
   RSTUVWXYZ[\]^_`
  abcdefghijklmnopq
rstuvwxyz{|}~
```

   Again we assume the ASCII character set.
5. Modify program ch1407 to break the text into phrases, using the comma and full stop as breaking characters. The output expected is given below.

```
The important issue about a language
is not so much what features the language possesses
but the features it does possess
are sufficient
to support the desired programming styles
in the desired application areas
```

6. Modify the above to break the text into words and count the frequency of occurrence of words by length. The output should be similar to that given below.

```
1   a              1
2   is so it to in              5
3   The not the but the are the the               8
4   much what does              3
5   issue about areas              3
6   styles              1
7   possess support desired desired               4
8   language features language features             4
9   important possesses              2
10  sufficient              1
11  programming application              2
```

7. 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.
8. 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.
9. 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 D (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?
10. 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 enciphered 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 enciphered 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 E contains an encoded message. Break it. Clue – Pg+Fybdujuvef jo Tdjfodf, Jorge Luis Borges.
11. Write a program that counts the total number of vowels in a sentence or text. Output the frequency of occurrence of each vowel.

Chapter 15
Complex

Make it as simple as possible, but no simpler.

Albert Einstein

Aims

The aims of this chapter are:

- To introduce the last predefined numeric data type in Fortran.
- To illustrate with examples how to use this type.

15.1 Introduction

This variable type reflects an extension of the real data type available in Fortran – the complex data type, where we can store and manipulate complex variables. Problems that require this data type are restricted to certain branches of mathematics, physics and engineering. 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 not supported in many programming languages as a base type which makes Fortran the language of first choice for many people.

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=cmplx(1.0,2.0)
```

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_15, © Springer-Verlag London Limited 2012

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 such as `log, sin, cos,` etc., accept a complex data type as argument.

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, `cmplx` and `int` can be used to 'force' any variable to real, complex or integer type.

There are a number of intrinsic functions to enable complex calculations to be performed. The program segment below uses some of them:

```
complex:: z, z1,z2,z3,zbar
real :: x,y,x1,y1,x2,y2,x3,y3,zmod
   z1 = cmplx (1.0, 2.0)      !  1 + i 2
   z2 = cmplx(x2, y2)         ! x2 + i y2
   z3 = cmplx (x3, y3)        ! x3 + i y3
   Z  = Z1*Z2 / Z3
   x  = real(z)               ! real part of z
   y  = aimag (z)             ! imaginary part of z
   zmod = abs(z)              ! modulus of z
   ZBAR = CONJG(Z)            ! complex conjugate of Z
```

15.2 Example 1

The second order differential equation:

$$\frac{d^2y}{dt^2} + 2\frac{dy}{dt} + y = x(t)$$

could describe the behaviour of an electrical system, where x(t) is the input voltage and y(t) is the output voltage and dy/dt is the current. The complex ratio

$$\frac{y(w)}{x(w)} = 1/(-w^2 + 2jw + 1)$$

is called the frequency response of the system because it describes the relationship between input and output for sinusoidal excitation at a frequency of w and where j

is $\sqrt{-1}$. The following program segment reads in a value of w and evaluates the frequency response for this value of w together with its polar form (magnitude and phase):

```
program ch1501
implicit none
!
! program to calculate frequency response of a system
! for a given Omega
! and its polar form (magnitude and phase).
!
real :: Omega ,real_part , Imag_part , Magnitude, Phase
complex:: Frequency_response
!
! Input frequency Omega
!
  print *, 'Input frequency'
  read *,Omega
!
  Frequency_response = 1.0 / &
    cmplx( - Omega * Omega + 1.0 , 2.0 * Omega)
  real_part = real(Frequency_response)
  Imag_part = aimag(Frequency_response)
!
! Calculate polar coordinates (magnitude and phase)
!
  magnitude = abs(frequency_response)
  phase = atan2 (imag_part, real_part)
!
  print *, ' at frequency ',omega
  print *, 'response = ', real_part,' + I ',imag_part
  print *, 'in polar form'
  print *, ' magnitude = ', magnitude
  print *, ' phase = ', phase
end program ch1501
```

15.3 Example 2

Here is a complete example of using some of the intrinsics with complex numbers.

```
program ch1502
implicit none
real:: x,y,x1,y1,x2,y2,x3,y3,zmod
complex:: z,z1,z2,z3,zbar
    print*,'input x1,y1,x2,y2,x3,y3'
    read*,x1,y1,x2,y2,x3,y3
    z1 = cmplx(1.0,2.0)
    z2 = cmplx(x2,y2)
    print*,'z2=',z2
    z3 = cmplx(x3-x1,(y3+y1)**2)
    print*,'z3=',z3
    z = z1*z2/z3
    x = real(z)
    y = aimag(z)
    zmod = abs(z)
    zbar = conjg(z)
    print*,'z =',z
    print*,'modulus of z =',zmod
    print*,'complex conjugate of z = ',zbar
end program ch1502
```

15.4 Complex and Kind Type

The standard requires that there be a minimum of two kind types for real numbers and this is also true of the complex data type. Chapter 5 must be consulted for a full coverage of real kind types. We would therefore use something like the following to select a complex kind type other than the default:

```
integer , parameter :: &
    Long_complex=selected_real_kind(15,307)
complex (Long_complex) :: Z
```

Chapter 21 includes a good example of how to use modules to define and use precision throughout a program and subprogram units.

15.5 Summary

Complex is used to store and manipulate complex numbers: those with a real and an imaginary part.

There are standard functions which allow conversion between the numerical data types – `cmplx, real` and `int`.

15.6 Problem

1. The program used in Chap. 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 take its square root, use the function `sqrt`. The formulae for the complex roots are

$$\frac{-b}{2a} \pm i\frac{\sqrt{-(b^2 - 4ac)}}{2a}$$

   If you manage this to your satisfaction, try your skills on the roots of a cubic (see the problems in Chap. 12).

Chapter 16
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 examine the last predefined type available in Fortran: logical.
- To introduce the concepts necessary to use logical expressions effectively, namely:
  - Logical variables.
  - Logical operators.
  - The hierarchy of operations.
  - Truth tables.

16.1 Introduction

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 in the following:

```
flag=.true.
```

or

```
flag=.false.
```

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_16, © Springer-Verlag London Limited 2012

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 == b) then
   .
else
   .
endif
```

The logical expression `(a == 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 two 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 intersection.
`.or.` logical union.

To illustrate the use of these operators, consider the following program extract:

```
logical :: a,b,c
   a=.true.
   b=.not.a      !(b now has the value 'false')
   c=a.or.b      !(c has the value 'true')
   c=a.and.b     !(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 than 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 (= =, >, <, >=, <= /=).
- `.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 significantly 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)then
      ...
      print *,'lots of printout'
      ...
   endif
```

ensures that, while debugging 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 the following:

```
logical :: up, down
   up=down+.false.
```

or

```
logical :: a2
    real dimension(10):: omega
    .
    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 and complex variable types are much more weakly typed (which helps lead to the confusion inherent in mixing variable types in arithmetic assignments).

16.2 I/O

Since logicals may take only the values `.true.` and `.false.`, the possibilities in reading and writing logical values are clearly limited. The L edit descriptor or format allows logicals to be input and output. On input, if the first nonblank characters are either T or .T, the logical value `.true.` is stored in the corresponding list item; if the first nonblank characters are F or .F, then `.false.` is stored. (Note therefore that reading, say, `ted` and `fahr` in an L4 format would be acceptable.) if the first nonblank 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 100, 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.

16.3 Summary

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.`
  - `.or.`
- Logical operators have a lower order of precedence than any others.

16.4 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 program 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.

Chapter 17
Introduction to Derived Types

Russell's theory of types leads to certain complexities in the foundations of mathematics… Its interesting features for our purposes are that types are used to prevent certain erroneous expressions from being used in logical and mathematical formulae; and that a check against violation of type constraints can be made purely by scanning the text, without any knowledge of the value which a particular symbol might happen to have.

C.A.R. Hoare, Structured Programming

Aims

The aim of this chapter is to introduce the concepts and ideas involved in using the facilities offered in Fortran 90 for the construction and use of user defined types:

- The way in which we define our own types.
- The way in which we declare variables to be of a user defined type.
- The way in which we manipulate variables of our own types.
- The way in which we can nest types within types.

The examples are simple and are designed to highlight the syntax. More complex and realistic examples of the use of user defined data types are to be found in later chapters.

17.1 Introduction

In the coverage so far we have used the intrinsic types provided by Fortran. The only data structuring technique available has been to construct arrays of these intrinsic types. Whilst this enables us to solve a reasonable variety of problems, it is

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_17, © Springer-Verlag London Limited 2012

inadequate for many purposes. In this chapter we look at the facilities offered by Fortran for the construction of our own types and how we manipulate data of these new, user defined types.

With the ability to define our own types we can now construct aggregate data types that have components of a variety of base types. These are given a variety of names including

- Record in the Pascal family of languages and in many older books on computing and data structuring;
- Structs in C;
- Classes in C++, Java, C# and Eiffel;
- Cartesian product is often used in mathematics and this is the terminology adopted by Hoare;

We will the term user defined type and derived types interchangeably.

There are two stages in the process of creating and using our own data types: we must first define the type, and then create variables of this type.

17.2 Example 1: Dates

```
program ch1701
implicit none
type date
   integer :: day=1
   integer :: month=1
   integer :: year=2000
end type date
type (date) :: d
   print *,d%day, d%month, d%year
   print *,'type in the date, day, month, year'
   read *,d%day, d%month, d%year
   print *,d%day, d%month, d%year
end program ch1701
```

This complete program illustrates both the definition and use of the type. It also shows how you can define initial values within the type definition.

17.3 Type Definition

The type *date* is defined to have three component parts, comprising a *day*, a *month* and a *year*, all of integer type. The syntax of a type construction comprises:

```
type Typename
  data type :: Component_name
  etc
end type Typename
```

Reference can then be made to this new type by the use of a single word, `date`, and we have a very powerful example of the use of abstraction.

17.4 Variable Definition

This is done by

```
type (Typename) :: Variablename
```

and we then define a variable D to be of this new type. The next thing we do is have a `read *` statement that prompts the user to type in three integer values, and the data are then echoed straight back to the user. We use the notation `Variablename% Component_Name` to refer to each component of the new data type.

17.4.1 Example 1 Variant Using Modules

The following is a variant on the above and achieves the same result with a small amount of additional syntax.

```
module date_module

   type date
     integer :: day=1
     integer :: month=1
     integer :: year=2000
   end type date

end module date_module

program ch1702

use date_module

implicit none

type (date) :: d

  print *,d%day, d%month, d%year
  print *,' type in the date, day, month, year'
  read *,d%day, d%month, d%year
  print *,d%day, d%month, d%year

end program ch1702
```

The key here is that we have embedded the type declaration inside a module, and then used the module in the main program.

If you are only using the type within one program unit then the first form is satisfactory, but if you are going to use the type in several program units the second is the required form.

We will use the second form in the examples that follow.

17.5 Example 2: Address Lists

```
module address_module

  type address
    character (len=40) :: name
    character (len=60) :: street
    character (len=60) :: district
    character (len=60) :: city
    character (len=8)  :: post_code
  end type address

end module address_module

program ch1703
use address_module
implicit none
integer  :: n_of_address
type (address) , dimension(:), allocatable:: addr
integer :: i
   print *,'input number of addresses'
   read *,n_of_address
   allocate(addr(1:n_of_address))
   open(unit=1,file="address.txt")
   do i=1,n_of_address
       read(unit=1,fmt='(a40)') addr(i)%name
       read(unit=1,fmt='(a60)') addr(i)%street
       read(unit=1,fmt='(a60)') addr(i)%district
       read(unit=1,fmt='(a60)') addr(i)%city
       read(unit=1,fmt='(a8)') addr(i)%post_code
   end do
   do i=1,n_of_address
       print *,addr(i)%name
       print *,addr(i)%street
```

```
      print *,addr(i)%district
      print *,addr(i)%city
      print *,addr(i)%post_code
   end do
end program ch1703
```

In this example we define a type `Address` which has components that one would expect for a person's address. We then define an array `Addr` of this type. Thus we are now creating arrays of our own user defined types. We index into the array in the way we would expect from our experience with integer, real and character arrays. The complete example is rather trivial in a sense in that the program merely reads from one file and prints the file out to the screen. However, it highlights many of the important ideas of the definition and use of user defined types.

17.6 Example 3: Nested User Defined Types

The following example builds on the two data types already introduced. Here we construct nested user defined data types based on them and construct a new data type containing them both plus additional information.

```
module personal_module

  type address
    character (len=60)  :: street
    character (len=60)  :: district
    character (len=60)  :: city
    character (len=8 )  :: post_code
  end type address

  type date_of_birth
    integer :: day
    integer :: month
    integer :: year
  end type date_of_birth

  type personal
    character (len=20)    :: first_name
    character (len=20)    :: other_names
    character (len=40)    :: surname
    type (date_of_birth) :: dob
    character (len=1)     :: sex
  type (address)         :: addr

end type personal

end module personal_module
```

```
program ch1704
use personal_module
implicit none
integer :: n_people
integer :: i
type (personal) , dimension(:), allocatable :: p

  print*,'input number of people'
  read*, n_people
  allocate(p(1:n_people))
  open(unit=1,file='person.txt')
  do i=1,n_people
    read(1,fmt=10)  p(i)%first_name,&
                 p(i)%other_names,&
                 p(i)%surname,&
                 p(i)%dob%day,&
                 p(i)%dob%month,&
                 p(i)%dob%year,&
                 p(i)%sex,&
                 p(i)%addr%street,&
                 p(i)%addr%district,&
                 p(i)%addr%city,&
                 p(i)%addr%post_code
                10 format(a20,/,&
                           a20,/,&
                           a40,/,&
                           i2,1x,i2,1x,i4,/,&
                           a1,/,&
                           a60,/,&
                           a60,/,&
                           a60,/,&
                           a8)
  end do
  do i=1,n_people
   write(*,fmt=20)   p(i)%first_name,&
                  p(i)%other_names,&
                  p(i)%surname,&
                  p(i)%dob%day,&
                  p(i)%dob%month,&
                  p(i)%dob%year,&
                  p(i)%sex,&
                  p(i)%addr%street,&
                  p(i)%addr%district,&
                  p(i)%addr%city,&
```

```
                  p(i)%addr%post_code
   20 format(        a20,a20,a40,/,&
                  i2,1x,i2,1x,i4,/,&
                  a1,/,&
                  a60,/,&
                  a60,/,&
                  a60,/,&
                  a8)
   end do
end program ch1704
```

Here we have a date of birth data type (`Date_Of_Birth`) based on the `Date` data type from the first example, plus a slightly modified address data type, incorporated into a new data type comprising personal details. Note the way in which we reference the component parts of this new, aggregate data type.

17.7 Problem

1. Modify the last example to include a more elegant printed name. The current example will pad with blanks the first name, other names and surname and span 80 characters on one line, which looks rather ugly.

Add a new variable name which will comprise all three subcomponents and write out this new variable, instead of the three subcomponents.

17.8 Bibliography

Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, London (1972)

This is one of the earliest and best introductions to data structures and structured programming. The whole book hangs together very well, and the section on data structures is a must for serious programmers.

Vowels, R.A.: Algorithms and Data Structures in F and Fortran. Unicomp, Tucson (1989)

One of the few books looking at algorithms and data structures using Fortran.

Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs (1976)

Wirth, N.: Algorithms + Data Structures. Prentice-Hall, Englewood Cliffs (1986)

The first is in Pascal, and the second in Modula 2.

Wood, D.: Paradigms and Programming in Pascal. Computer Science Press, Rockville (1984)

contains a number of examples of the use of recursion in problem solving. Also provides a number of useful case studies in problem solving.

Chapter 18
An Introduction to Pointers

Not to put too fine a point on it.

Charles Dickens, Bleak House.

Aim

The primary aim of the chapter is to introduce some of the key concepts of pointers in Fortran.

18.1 Introduction

All of the data types introduced so far, with the exception of the allocatable array, have been static. Even with the allocatable array a size has to be set at some stage during program execution. The facilities provided in Fortran by the concept of a pointer combined with those offered by a user defined type enable us to address a completely new problem area, previously extremely difficult to solve in Fortran. There are many problems where one genuinely does not know what requirements there are on the size of a data structure. Linked lists allow sparse matrix problems to be solved with minimal storage requirements, two-dimensional spatial problems can be addressed with quad-trees and three-dimensional spatial problems can be addressed with oct-trees. Many problems also have an irregular nature, and pointer arrays address this problem.

First we need to cover some of the technical aspects of pointers. A pointer is a variable that has the pointer attribute. A pointer is associated with a target by allocation or pointer assignment. A pointer becomes associated as follows:

- The pointer is allocated as the result of the successful execution of an allocate statement referencing the pointer

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_18, © Springer-Verlag London Limited 2012

or

- The pointer is pointer-assigned to a target that is associated or is specified with the target attribute and, if allocatable, is currently allocated.

A pointer shall neither be referenced nor defined until it is associated. A pointer is disassociated following execution of a deallocate or nullify statement, following pointer association with a disassociated pointer, or initially through pointer initialisation.

A pointer may have a pointer association status of associated, disassociated, or undefined. Its association status may change during execution of a program. Unless a pointer is initialised (explicitly or by default), it has an initial association status of undefined. A pointer may be initialised to have an association status of disassociated.

Let us look at some examples to clarify these points.

18.2 Some Basic Pointer Concepts

With the introduction of pointers as a data type into Fortran we also have the introduction of a new assignment statement – the pointer assignment statement. Consider the following example:

```
program ch1801
implicit none
integer , pointer :: a=>null(),b=>null()
integer , target :: c
integer :: d
  c = 1
  a => c
  c = 2
  b => c
  d = a + b
  print *,a,b,c,d
end program ch1801
```

The following

```
integer , pointer :: a=>null(),b=>null()
```

is a declaration statement that defines `a` and `b` to be variables, with the pointer attribute. This means we can use `a` and `b` to refer or point to integer values. We also use the `null` intrinsic to set the status of the pointers `a` and `b` to disassociated. Using the `null` intrinsic means that we can test the status of a pointer variable and avoid making a number of common pointer programming errors. Note that in this case no space is set aside for the pointer variables `a` and `b`, i.e. `a` and `b` should not be referenced in this state.

The second declaration defines c to be an integer, with the target attribute, i.e., we can use pointers to refer or point to the value of the variable c.

The last declaration defines d to be an ordinary integer variable.

In the case of the last two declarations space is set aside to hold two integers.

Let us now look at the various executable statements in the program, one at a time:

`c = 1`	This is an example of the normal assignment statement with which we are already familiar. We use the variable name c in our program and whenever we use that name we get the value of the variable c
`a => c`	This is an example of a pointer assignment statement. This means that both a and c now refer to the same value, in this case 1. a becomes associated with the target c. a can now be referenced
`c = 2`	Conventional assignment statement, and c now has the value 2
`b = > c`	Second example of pointer assignment. b now points to the value that c has, in this case 2. b becomes associated with the target c. b can now be referenced
`d = a + b`	Simple arithmetic assignment statement. The value that a points to is added to the value that b points to and the result is assigned to d

The last statement prints out the values of a, b, c and d.

The output is

```
 2  2  2  4
```

18.3 The associated Intrinsic Function

The associated intrinsic returns the association status of a pointer variable. Consider the following example which is a simple variant on the first.

```
program ch1802
integer , pointer :: a=>null(),b=>null()
integer , target :: c
integer :: d
  print *,associated(a)
  print *,associated(b)
  c = 1
  a => c
  c = 2
  b => c
  d = a + b
  print *,a,b,c,d
  print *,associated(a)
  print *,associated(b)
end program ch1802
```

The output from running this program is shown below

```
 F
 F
 2 2 2 4
 T
 T
```

and as you can see we therefore have a mechanism to test pointers to see if they are in a valid state before use.

18.4 Referencing `a` and `b` Before Allocation or Pointer Assignment

Consider the following example:

```
program ch1803
integer , pointer :: a=>null(),b=>null()
integer , target :: c
integer :: d
  print *,a
  print *,b
  c = 1
  a => c
  c = 2
  b => c
  d = a + b
  print *,a,b,c,d
end program ch1803
```

Here we are actually referencing the pointers a and b, even though their status is disassociated. Most compilers generate a run time error with this example with the default compiler options, and the error message tends to be a little cryptic. It is recommended that you look at the diagnostic compilation switches for you compiler. We include some sample output below from gfortran, Intel and Nag. The error messages are now much more meaningful.

18.4.1 gfortran

Switches are

```
gfortran -W -Wall -fbounds-check -pedantic-errors
-std=f2003 -Wunderflow -O -fbacktrace
-ffpe-trap=zero,overflow,underflow -g
```

The program runs to completion with no error message. Here is the output.

```
d:\document\fortran\newbook\examples\ch18>ch1803.out
             0
             0
             2             2              2             4
```

18.4.2 Intel

Switches are

```
/check:all /traceback
```

Here is the output.

```
D:\document\fortran\newbook\examples\ch18>ch1803
forrtl: severe (408): fort: (7): Attempt to use pointer
A when it is not associated with a target
Image                PC                  Routine
Line         Source
ch1803.exe          000000013F0AC598  Unknown
Unknown  Unknown
...
ntdll.dll           0000000077096611  Unknown
Unknown  Unknown
```

18.4.3 Nag

Switches are

```
-C=all -C=undefined -info -g -gline
```

Here is the output.

```
Runtime Error: ch1803.f90, line 5: Reference to
disassociated POINTER A
Program terminated by fatal error
ch1803.f90, line 5: Error occurred in CH1803
```

18.5 Pointer Allocation and Assignment

Consider the following example:

```
program ch1804
integer , pointer :: a=>null(),b=>null()
```

```
integer , target :: c
integer :: d
   allocate(a)
   a = 1
   c = 2
   b => c
   d = a + b
   print *,a,b,c,d
   deallocate(a)
end program ch1804
```

In this example we allocate a and then can do conventional assignment. If we had not allocated a the assignment would be illegal. Try out problem 2 to see what will happen with your compiler.

Our simple recommendation when using pointers is to nullify them when declaring them and to explicitly allocate them before conventional assignment.

18.6 Memory Leak Examples

Dynamic memory brings greater versatility but requires greater responsibility. Consider the following example:

```
program ch1805
integer , pointer :: a=>null(),b=>null()
integer , target :: c
integer :: d
   allocate(a)
   allocate(b)
   a=100
   b=200
   print *,a,b
   c = 1
   a => c
   c = 2
   b => c
   d = a + b
   print *,a,b,c,d
end program ch1805
```

What has happened to the memory allocated to `a` and `b`?

Now consider the following example.

```
program ch1806
implicit none
integer :: allocate_status=0
integer , parameter :: n1=10000000
integer , parameter :: n2=5
integer , dimension(:)     , pointer :: x
integer , dimension(1:n2) , target :: y
integer :: i
  do
    allocate(x(1:n1),stat=allocate_status)
    if (allocate_status > 0) then
      print *,' allocate failed. program ends.'
      stop
    endif
    do i=1,n1
      x(i)=I
    end do
    do i=1,n2
      print *,x(i)
    end do
    do i=1,n2
      y(i)=i*I
    end do
    do i=1,n2
      print *,y(i)
    end do
! x now points to y
    x=>y            ! x now points to y
    do i=1,n2
      print *,x(i)
    end do
! what has happened to the memory that x
! used to point to?
  end do
end program ch1806
```

Before running the above example we recommend starting up a memory monitoring program.

Under Microsoft Windows XP Professional holding [CTRL] + [ALT] + [DEL] will bring up the Windows Task Manager. Choose the [Performance] tab to get a screen which will show CPU usage, PF Usage, CPU Usage History and Page File Usage History. You will also get details of Physical and Kernel memory usage.

Under Linux type

```
top
```

in a terminal window.

In these examples we also see the recommended form of the allocate statement when working with arrays. This enables us to test if the allocation has worked and take action accordingly. A positive value indicates an allocation error, zero indicates OK.

The second program can require a power off on a Windows operating system with a compiler that will remain anonymous!

18.7 Non-standard Pointer Example

Some Fortran compilers provide a non-standard `loc` intrinsic. This can be used to print out the address of the variable passed as an argument. Here is the program.

```
program ch1807
integer , pointer :: a=>null(),b=>null()
integer , target :: c
integer :: d
   allocate(a)
   allocate(b)
   a=100
   b=200
   print *,a,b
   print *,loc(a)
   print *,loc(b)
   print *,loc(c)
   print *,loc(d)
   c = 1
   a => c
   c = 2
   b => c
   d = a + b
   print *,a,b,c,d
   print *,loc(a)
   print *,loc(b)
   print *,loc(c)
   print *,loc(d)
end program ch1807
```

Here is the output from a compiler with `loc` support.

```
 100          200
      13803552
      13803600
       2948080
       2948084
   2             2            2              4
       2948080
       2948080
       2948080
       2948084
```

This program clearly shows the memory leak.

18.8 Problems

1. Compile and run all of the example programs in this chapter with your compiler and examine the output.
2. Compile and run example 4 without the `allocate(a)` statement. See what happens with your compiler.

Here is the output from the Nag compiler. The first run is with the default options.

```
nagfor ch1804p.f90
NAG Fortran Compiler: Release 5.2(722)
[NAG Fortran Compiler normal termination]
a.exe
```

There is no meaningful output.
The following adds the -C=all compilation option.

```
nagfor ch1804p.f90 -C=all
NAG Fortran Compiler: Release 5.2(722)
[NAG Fortran Compiler normal termination]
a.exe
Runtime Error: ch1804p.f90, line 5: Reference to
disassociated POINTER A
Program terminated by fatal error
```

We now get a meaningful error message.

Chapter 19
Introduction to Subroutines

A man should keep his brain attic stacked with all the furniture he is likely to use, and the rest he can put away in the lumber room of his library, where he can get at it if he wants.

Sir Arthur Conan Doyle, Five Orange Pips

Aims

The aims of this chapter are:

- To consider some of the reasons for the inclusion of subroutines in a programming language.
- To introduce with a concrete example some of the concepts and ideas involved with the definition and use of subroutines.
  - Arguments or parameters.
  - The intent attribute for parameters.
  - The call statement.
  - Scope of variables.
  - Local variables and the save attribute.
  - The use of parameters to report on the status of the action carried out in the subroutine.
- Module procedures to provide interfaces.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_19, © Springer-Verlag London Limited 2012

19.1 Introduction

In the earlier chapter on functions we introduced two types of function

- Intrinsic functions – which are part of the language.
- User defined functions – by which we extend the language.

We now introduce subroutines which collectively with functions are given the name procedures.
Procedures provide a very powerful extension to the language by:

- Providing us with the ability to break problems down into simpler more easily solvable subproblems.
- Allowing us to concentrate on one aspect of a problem at a time.
- Avoiding duplication of code.
- Hiding away messy code so that a main program is a sequence of calls to procedures.
- Providing us with the ability to put together collections of procedures that solve commonly occurring subproblems, often given the name libraries, and generally compiled.
- Allowing us to call procedures from libraries written, tested and documented by experts in a particular field. There is no point in reinventing the wheel!

There are a number of concepts required for the successful use of subroutines and we met some of them in Chap. 12 when we looked at user defined functions. We will extend the ideas introduced there of parameters and introduce the additional concept of an interface via the use of modules. The ideas are best explained with a concrete example.

Note that we use the terms parameters and arguments interchangeably.

19.2 Example 1

This example is one we met earlier that solves a quadratic equation, i.e., solves

$$ax^2 + bx + c = 0$$

The program to do this originally was just one program. In the example below we break that problem down into smaller parts and make each part a subroutine. The components are:

- Main program or driving routine.
- Interaction with user to get the coefficients of the equation.
- Solution of the quadratic.

Let us look now at how we do this with the use of subroutines:

```
module interact_module
contains
subroutine interact(a,b,c,ok)
   implicit none
   real , intent(out) :: a
   real , intent(out) :: b
   real , intent(out) :: c
   logical , intent(out) :: ok
   integer :: io_status=0
   print*,' type in the coefficients a, b and c'
   read(unit=*,fmt=*,iostat=io_status)a,b,c
   if (io_status == 0) then
     ok=.true.
   else
     ok=.false.
   endif
end subroutine interact
end module interact_module

module solve_module
contains
subroutine solve(e,f,g,root1,root2,ifail)
   implicit none
   real , intent(in) :: e
   real , intent(in) :: f
   real , intent(in) :: g
   real , intent(out) :: root1
   real , intent(out) :: root2
   integer , intent(inout) :: ifail
!  local variables
   real :: term
   real :: a2
   term = f*f - 4.*e*g
   a2 = e*2.0
!  if term < 0, roots are complex
   if(term < 0.0)then
      ifail=1
   else
     term = sqrt(term)
     root1 = (-f+term)/a2
     root2 = (-f-term)/a2
   endif
end subroutine solve
end module solve_module
```

```
program ch1901
use interact_module
use solve_module
implicit none
! simple example of the use of a main program and two
! subroutines. one interacts with the user and the
! second solves a quadratic equation,
! based on the user input.

real :: p, q, r, root1, root2
integer :: ifail=0
logical :: ok=.true.
   call interact(p,q,r,ok)
   if (ok) then
     call solve(p,q,r,root1,root2,ifail)
     if (ifail == 1) then
       print *,' complex roots'
       print *,' calculation abandoned'
     else
       print *,' roots are   ',root1,'   ',root2
     endif
   else
     print*,' error in data input program ends'
   endif
end program ch1901
```

19.2.1 Defining a Subroutine

A subroutine is defined as

subroutine subroutine_name (optional list of dummy arguments)
implicit none
dummy argument type definitions with intent
…
end subroutine subroutine_name

and from the earlier example we have the subroutine

```
subroutine interact(a,b,c,ok)
   implicit none
   real , intent(out) :: a
   real , intent(out) :: b
   real , intent(out) :: c
   logical , intent(out) :: ok
   integer :: io_status=0
```

```
   print*,' type in the  coefficients  a,  b and c'
   read(unit=*,fmt=*,iostat=io_status)a,b,c
   if (io_status ==  0)  then
     ok=.true.
   else
     ok=.false.
   endif
end subroutine interact
```

19.2.2 Referencing a Subroutine

To reference a subroutine you use the `call` statement:

call subroutine_name(optional list of actual arguments)

and from the earlier example the call to subroutine interact was of the form:

```
call interact(p,q,r,ok)
```

When a subroutine returns to the calling program unit control is passed to the statement following the `call` statement.

19.2.3 Dummy Arguments or Parameters and Actual Arguments

Procedures and their calling program units communicate through their arguments. We often use the terms parameter and arguments interchangeably throughout this text. The `subroutine` statement normally contains a list of dummy arguments, separated by commas and enclosed in brackets. The dummy arguments have a type associated with them; for example, in subroutine solve x is of type `real`, but no space is put aside for this in memory. When the subroutine is referenced e.g., `call` solve(p,q,r,root1,root2,ifail), then the dummy argument points to the actual argument p, which is a variable in the calling program unit. The dummy argument and the actual argument must be of the same type – in this case `real`.

19.2.4 Intent

It is recommended that dummy arguments have an `intent` attribute. In the earlier example subroutine `solve` has a dummy argument `e` with `intent(in)`, which means that when the subroutine is referenced or called it is expecting `e` to have a value, but its value cannot be changed inside the subroutine. This acts as an extra security measure besides making the program easier to understand. For each parameter it may have one of three attributes:

- `intent(in)`, where the parameter already has a value and cannot be altered in the called routine.
- `intent(out)`, where the parameter does not have a value, and is given one in the called routine.
- `intent(inout)`, where the parameter already has a value and this is changed in the called routine.

19.2.5 Local Variables

We saw with functions that variables could be essentially local to the function and un available elsewhere. The concept of local variables also applies to subroutines. In the example above term and a2 are both local variables to the subroutine `solve`.

19.2.6 Local Variables and the Save Attribute

Local variables are usually created when a procedure is called and their value lost when execution returns to the calling program unit. To make sure that a local variable retains its values between calls to a subprogram the save attribute can be used on a type statement; e.g.,

```
integer , save :: I
```

means that when this statement appears in a subprogram the value of the local variable `I` is saved between calls.

19.2.7 Scope of Variables

In most cases variables are only available within the program unit that defines them. The introduction of argument lists to procedures immediately opens up the possibility of data within one program unit becoming available in one or more other program units.

In the main program we declare the variables `p, q, r, root1, root2, ifail` and `ok`.

Subroutine `interact` has one local variable io_status. It works on the arguments `a, b, c` and `ok`; which map onto `p, q, r` and `ok` from the main program.

Subroutine solve has two locally defined variables, `term` and `a2`. It works with the variables `e, f, g, root1, root2` and `ifail`, which map onto `p, q, r, root1, root2` and `ifail` from the main program.

19.2.8 Status of the Action Carried Out in the Subroutine

It is also useful to use parameters that carry information regarding the status of the action carried out by the subroutine. With the subroutine `interact` we use a logical variable `ok` to report on the status of the interaction with the user. In the subroutine `solve` we use the status of the integer variable `ifail` to report on the status of the solution of the equation.

19.2.9 Modules 'containing' Procedures

At the same time as introducing procedures we have 'contained' them in a module and then the main program 'uses' the module in order to make the procedure available. Procedures 'contained' in modules are called module procedures.

With the 'use' statement the interface to the procedure is available to the compiler so that the types and positions of the actual and dummy arguments can be checked. This was a major source of errors with Fortran 77.

The `use` statement must be the first statement in the main program or calling unit, also the modules must be compiled before the program or calling unit.

We will cover modules in more depth in later chapters.

There are times when an interface is mandatory in Fortran so it's good practice to use module procedures from the start. There are other ways of providing explicit interfaces and we will cover them later.

19.3 Why Bother with Subroutines?

Given the increase in the complexity of the overall program to solve a relatively straightforward problem, one must ask why bother. The answer lies in our ability to manage the solution of larger and larger problems. We need all the help we can get if we are to succeed in our task of developing large-scale reliable programs.

We need to be able to break our problems down into manageable subcomponents and solve each in turn. We are now in a very good position to be able to do this. Given a problem that requires a main program, one or more functions and one or more subroutines we can work on each subcomponent in relative isolation, and know that by using features like module procedures we will be able to glue all of the components together into a stable structure at the end. We can independently compile the main program and the modules containing the functions and subroutines and use the linker to generate the overall executable, and then test that.

Providing we keep our interfaces the same we can alter the actual implementations of the functions and subroutines and just recompile the changed procedures.

19.4 Summary

We now have the following concepts for the use of subroutines:

- Module procedures providing interfaces.
- Intent attribute for parameters.
- Dummy parameters.
- The use of the call statement to invoke a subroutine.
- The concepts of variables that are local to the called routines and are unavailable elsewhere in the overall program.
- Communication between program units via the argument list.
- The concept of parameters on the call that enable us to report back on the status of the called routine.

19.5 Problems

1. Type the program and module procedures for example 1 into one file. Compile, link and run providing data for complex roots to test this part of the code.
2. Split the main program and modules up into three separate files. Compile the modules and then compile the main program and link the object files to create one executable. Look at the file size of the executable and the individual object files. What do you notice?
   The development of large programs is eased considerably by the ability to compile small program units and eradicate the compilation errors from one unit at a time. The linker obviously also has an important role to play in the development process.
3. Write a subroutine to calculate new coordinates (x' , y') from (x, y) when the axes are rotated counter clockwise through an angle of a radians using:

$$x' = x\cos a + y\sin a$$
$$y' = -x\sin a + y\cos a$$

Hint:

The subroutine would look something like
subroutine ChangeCoordinate(x, y, a, xd, yd)
Write a main program to read in values of x, y and a and then call the subroutine and print out the new coordinates. Use a module procedure.

Chapter 20
Subroutines: 2

It is one thing to show a man he is in error, and another to put him in possession of the truth.

John Locke

Aims

The aims of this chapter are to extend the ideas in the earlier chapter on subroutines and look in more depth at parameter passing, in particular using a variety of ways of passing arrays.

20.1 More on Parameter Passing

So far we have seen scalar parameters of type real, integer and logical. We will now look at numeric array parameters and character parameters. We need to introduce some technical terminology first. Don't panic if you don't fully understand the terminology as the examples should clarify things.

20.1.1 Assumed-Shape Array

An assumed-shape array is a nonpointer dummy argument array that takes its shape from the associated actual argument array.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_20, © Springer-Verlag London Limited 2012

20.1.2 Deferred-Shape Array

A deferred-shape array is an allocatable array or an array pointer. An allocatable array is an array that has the allocatable attribute and a specified rank, but its bounds, and hence shape, are determined by allocation or argument association.

20.1.3 Automatic Arrays

An automatic array is an explicit-shape array that is a local variable. Automatic arrays are only allowed in function and subroutine subprograms, and are declared in the specification part of the subprogram. At least one bound of an automatic array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

20.2 Example 1 – Assumed Shape Parameter Passing

We are going to use an example based on a main program and a subroutine that calculates the mean and standard deviation of an array of numbers. The subroutine has the following parameters:

- x – the array containing the real numbers.
- n – the number of elements in the array.
- mean – the mean of the numbers.
- std_dev – the standard deviation of the numbers.

Consider the following program and subroutine.

```
module statistics_module
implicit none

contains
   subroutine stats(x,n,mean,std_dev)
   implicit none
   integer, intent (in) :: n
   real, intent (in), dimension (:) :: x
   real, intent (out) :: mean
   real, intent (out) :: std_dev
   real :: variance
   real :: sumxi, sumxi2
   integer :: i

     variance = 0.0
     sumxi = 0.0
     sumxi2 = 0.0
     do i = 1, n
```

```
      sumxi = sumxi + x(i)
      sumxi2 = sumxi2 + x(i)*x(i)
    end do
    mean = sumxi/n
    variance = (sumxi2-sumxi*sumxi/n)/(n-1)
    std_dev = sqrt(variance)
  end subroutine stats
end module statistics_module

program ch2001
use statistics_module
implicit none
integer, parameter :: n = 10
real, dimension (1:n) :: x
real, dimension (-4:5) :: y
real, dimension (10)  :: z
real, allocatable, dimension (:) :: t
real :: m, sd
integer :: i

  do i = 1, n
    x(i) = real(i)
  end do
  call stats(x,n,m,sd)
  print *, ' x'
  print *, ' Mean = ', m
  print *, ' Standard deviation = ', sd
  y = x
  call stats(y,n,m,sd)
  print *, ' y'
  print *, ' Mean = ', m
  print *, ' Standard deviation = ', sd
  z = x
  call stats(z,10,m,sd)
  print *, ' z'
  print *, ' Mean = ', m
  print *, ' Standard deviation = ', sd
  allocate (t(n))
  t = x
  call stats(t,10,m,sd)
  print *, ' t'
  print *, ' Mean = ', m
  print *, ' Standard deviation = ', sd
end program ch2001
```

A fundamental rule in modern Fortran is that the shape of an actual array argument and its associated dummy arguments are the same, i.e., they both must have

the same rank and the same extents in each dimension. The best way to apply this rule is to use assumed-shape dummy array arguments as shown in the example above.

In the subroutine we have

```
real , intent(in) , dimension(:) :: x
```

where x is an assumed-shape dummy array argument, and it will assume the shape of the actual argument when the subroutine is called.

In two of the calls we have passed a variable `n` as the size of the array and used a literal integer constant (10) in the other two cases. Both parameter passing mechanisms work.

20.2.1 Notes

There are several restrictions when using assumed-shape arrays:

- The rank is equal to the number of colons, in this case 1.
- The lower bounds of the assumed-shape array are the specified lower bounds, if present, and 1 otherwise. In the example above it is 1 because we haven't specified a lower bound.
- The upper bounds will be determined on entry to the procedure and will be whatever values are needed to make sure that the extents along each dimension of the dummy argument are the same as the actual argument. In this case the upper bound will be n.
- An assumed-shape array must not be defined with the pointer or allocatable attribute in Fortran.
- When using an assumed-shape array an interface is mandatory. In this example it is provided by the stats subroutine being a contained subroutine in a module, and the use of the module in the main program.

20.3 Character Arguments and Assumed-Length Dummy Arguments

The types of parameters considered so far have been `real`, `integer` and `logical`. Character variables are slightly different because they have a length associated with them. Consider the following program and subroutine which, given the name of a file, opens it and reads values into the `real` array `x`:

```
module read_module
implicit none
contains
subroutine readin(name,x,n)
implicit none
```

```
integer , intent(in) :: n
real,dimension(:),intent(out)::x
character (len=*),intent(in)::name
integer::I
  open(unit=10,status='old',file=name)
  do i=1,n
    read(10,*)x(i)
  end do
  close(unit=10)
end subroutine readin
end module read_module

program ch2002
use read_module
implicit none
real , allocatable , dimension(:) :: a
integer :: nos,i
character(len=20)::filename

  print *,' Type in the name of the data file'
  read '(a)' , filename
  print *,' Input the number of items'
  read * , nos
  allocate (a(1:nos))
  call readin(filename,a,nos)
  print * , ' data read in was'
  do i=1,nos
    print *,' ',a(i)
  enddo
end program ch2002
```

The main program reads the file name from the user and passes it to the subroutine that reads in the data. The dummy argument `name` is of type assumed-length, and picks up the length from the actual argument `filename` in the calling routine, which is in this case 20 characters. An interface must be provided with assumed-shape dummy arguments, and this is achieved in this case by the subroutine being in a module.

20.4 Rank 2 and Higher Arrays as Parameters

The following example illustrates the modern way of passing rank 2 and higher arrays as parameters. We start with a simple rank 2 example.

```
module matrix_module
   implicit none
```

```
contains
   subroutine matrix_bits(a,b,c,a_t,n)
     implicit none
     integer, intent (in) :: n
     real, dimension (:,:), intent (in) :: a, b
     real, dimension (:,:), intent (out) :: c, a_t
     integer :: i, j, k
     real :: temp

! matrix multiplication c=ab

     do i = 1, n
       do j = 1, n
         temp = 0.0
         do k = 1, n
           temp = temp + a(i,k)*b(k,j)
         end do
         c(i,j) = temp
       end do
     end do

! calculate a_t transpose of a

! set a_t to be transpose matrix a
     do i = 1, n
       do j = 1, n
         a_t(i,j) = a(j,i)
       end do
     end do
   end subroutine matrix_bits
end module matrix_module

program ch2003
  use matrix_module
  implicit none
  real, allocatable, dimension (:,:) :: &
    one, two, three, one_t
  integer :: i, n

  print *, 'input size of matrices'
  read *, n
  allocate (one(1:n,1:n))
  allocate (two(1:n,1:n))
  allocate (three(1:n,1:n))
  allocate (one_t(1:n,1:n))
```

```
  do i = 1, n
    print *, 'input row ', i, ' of one'
    read *, one(i,1:n)
  end do
  do i = 1, n
    print *, 'input row ', i, ' of two'
    read *, two(i,1:n)
  end do
  call matrix_bits(one,two,three,one_t,n)
  print *, ' matrix three:'
  do i = 1, n
    print *, three(i,1:n)
  end do
  print *, ' matrix one_t:'
  do i = 1, n
    print *, one_t(i,1:n)
  end do
end program ch2003
```

The subroutine is doing a matrix multiplication and transpose. There are intrinsic functions in Fortran called `matmul` and `transpose` that provide the same functionality as the subroutine. One of the problems at the end of the chapter is to replace the code in the subroutine with calls to the intrinsic functions.

20.4.1 Notes

The dummy array and actual array arguments look the same but there is a difference:

- The dummy array arguments `a, b, c, a_t` are all assumed-shape arrays and take the shape of the actual array arguments `one, two, three` and `one_t`, respectively.
- The actual array arguments `one, two, three` and `one_t` in the main program are allocatable arrays or deferred-shape arrays. An allocatable array is an array that has an allocatable attribute. Its bounds and shape are declared when the array is allocated, hence deferred-shape.

20.5 Automatic Arrays and Median Calculation

This example looks at the calculation of the median of a set of numbers and also illustrates the use of an automatic array.

The median is the middle value of a list, i.e., the smallest number such that at least half the numbers in the list are no greater. if the list has an odd number of entries, the median is the middle entry in the list after sorting the list into ascending

order, if the list has an even number of entries, the median is equal to the sum of the two middle (after sorting) numbers divided by two. One way to determine the median computationally is to sort the numbers and choose the item in the middle.

Wirth classifies sorting into simple and advanced, and his three simple methods are as follows:

- Insertion sorting – The items are considered one at a time and each new item is inserted into the appropriate position relative to the previously sorted item. If you have ever played bridge then you have probably used this method.
- Selection sorting – First the smallest (or largest) item is chosen and is set aside from the rest. Then the process is repeated for the next smallest item and set aside in the next position. This process is repeated until all items are sorted.
- Exchange sorting – if two items are found to be out of order they are interchanged. This process is repeated until no more exchanges take place.

Knuth also identifies the above three sorting methods. For more information on sorting the Knuth and Wirth books are good starting places. Knuth is a little old (1973) compared to Wirth (1986), but it is still a very good coverage. Knuth uses mix assembler to code the examples whilst the Wirth book uses Modula 2, and is therefore easier to translate into modern Fortran.

In the example below we use an exchange sort:

```
module statistics_module
   implicit none

contains
   subroutine stats(x,n,mean,std_dev,median)
     implicit none
     integer, intent (in) :: n
     real, intent (in), dimension (:) :: x
     real, intent (out) :: mean
     real, intent (out) :: std_dev
     real, intent (out) :: median
     real, dimension (1:n) :: y
     real :: variance
     real :: sumxi, sumxi2
     integer :: k

     sumxi = 0.0
     sumxi2 = 0.0
     variance = 0.0
     sumxi = sum(x)
     sumxi2 = sum(x*x)
     mean = sumxi/n
     variance = (sumxi2-sumxi*sumxi/n)/(n-1)
     std_dev = sqrt(variance)
     y = x
     if (mod(n,2)==0) then
```

```
      median = (find(n/2)+find((n/2)+1))/2
    else
      median = find((n/2)+1)
    end if

  contains
    real function find(k)
      implicit none
      integer, intent (in) :: k
      integer :: l, r, i, j
      real :: t1, t2

      l = 1
      r = n
      do while (l<r)
        t1 = y(k)
        I = l
        j = r
        do
          do while (y(i)<t1)
            I = I + 1
          end do
          do while (t1<y(j))
            j = j - 1
          end do
          if (i<=j) then
            t2 = y(i)
            y(i) = y(j)
            y(j) = t2
            I = I + 1
            j = j - 1
          end if
          if (i>j) exit
        end do
        if (j<k) then
          l = I
        end if
        if (k<i) then
          r = j
        end if
      end do
      find = y(k)
    end function find

  end subroutine stats
end module statistics_module
```

```
program ch2004
  use statistics_module
  implicit none
  integer :: n
  integer :: i
  real, allocatable, dimension (:) :: x
  real :: m, sd, median
  integer, dimension (8) :: timing

  n = 1000000
  do i = 1, 3
    print *, ' n = ', n
    allocate (x(1:n))
    call random_number(x)
    x = x*1000
    call date_and_time(values=timing)
    print *, ' initial '
    print *, timing(6), timing(7), timing (8)
    call stats(x,n,m,sd,median)
    print *, ' Mean = ', m
    print *, ' Standard deviation = ', sd
    print *, ' Median is = ', median
    call date_and_time(values=timing)
    print *, timing(6), timing(7), timing(8)
    n = n*10
    deallocate (x)
  end do

end program ch2004
```

In the subroutine stats the array y is automatic. It will be allocated automatically when we call the subroutine. We use this array as a work array to hold the sorted data. We then use this sorted array to determine the median.

Note the use of the `sum` intrinsic in this example:

```
sumxi=sum(x)
sumxi2=sum(x*x)
```

These statements replace the `do` loop from the earlier example. A good optimising compiler would not make two passes over the data with these two statements.

20.5.1 Internal Subroutines and Scope

The stats subroutine contains the find subroutine. The stats subroutine has access to the following variables

- x,n,mean,std_dev, median — these are made available as they are passed in as parameters.
- y, variance, sumxi, sumxi2 — are local to the subroutine stats.

The subroutine find has access to the above as it is contained within subroutine stats. It also has the following local variables that are only available within subroutine selection

- i,j,k, minimum

This program uses an algorithm developed by Hoare to determine the median. The number of computations required to find the median is approximately 2 * n.

The limiting factor with this algorithm on these systems is the amount of installed memory. The program crashes on both systems with a failure to allocate the automatic array. This is a drawback of automatic arrays in that there is no mechanism to handle this failure gracefully. You would then need to use allocatable local work arrays. The drawback here is that the programmer is then responsible for the deallocation of these arrays. Memory leaks are then possible.

20.6 Recursive Subroutines – Quicksort

In Chap. 12 we saw an example of recursive functions. This example illustrates the use of recursive subroutines. It uses a simple implementation of Hoare's Quicksort. References are given in the bibliography. The overall problem is broken down into:

- A main program that prompts the user for the name of the data file and n. The allocation of the array is carried out in the main program.
- A subroutine to read the data.
- A subroutine to sort the data. This subroutine contains the recursive sub routine Quicksort.
- A subroutine to write the sorted data to a file.

Below is the complete program:

```
module read_data_module
  implicit none

contains
  subroutine read_data(file_name,raw_data,how_many)
     implicit none
     character (len=*), intent (in) :: file_name
     integer, intent (in) :: how_many
     real, intent (out), dimension (:) :: raw_data
! local variables
     integer :: i
```

```
     open (file=file_name,unit=1)
     do i=1, how_many
       read (unit=1,fmt=*) raw_data(i)
     end do

  end subroutine read_data
end module read_data_module

module sort_data_module
  implicit none

contains
  subroutine sort_data(raw_data,how_many)
    implicit none
    integer, intent (in) :: how_many
    real, intent (inout), dimension (:) :: raw_data

    call quicksort(1,how_many)

  contains

    recursive subroutine quicksort(l,r)
      implicit none
      integer, intent (in) :: l, r
! local variables
      integer :: i, j
      real :: v, t

      I = l
      j = r
      v = raw_data(int((l+r)/2))
      do
        do while (raw_data(i)<v)
          I = I + 1
        end do
        do while (v<raw_data(j))
          j = j - 1
        end do
        if (i<=j) then
          t = raw_data(i)
          raw_data(i) = raw_data(j)
          raw_data(j) = t
          I = I + 1
          j = j - 1
        end if
        if (i>j) exit
      end do
```

```
        if (l<j) then
          call quicksort(l,j)
        end if

        if (i<r) then
          call quicksort(i,r)
        end if

      end subroutine quicksort

   end subroutine sort_data
end module sort_data_module

module print_data_module
   implicit none

contains
   subroutine print_data(raw_data,how_many)
      implicit none
      integer, intent (in) :: how_many
      real, intent (in), dimension (:) :: raw_data
! local variables
      integer :: i

      open (file='sorted.txt',unit=2)
      do i = 1, how_many
        write (unit=2,fmt=*) raw_data(i)
      end do
      close (2)
   end subroutine print_data

end module print_data_module

program ch2005
   use read_data_module
   use sort_data_module
   use print_data_module

   implicit none
   integer :: how_many
   character (len=20) ::  file_name
   real, allocatable, dimension (:) :: raw_data
   integer, dimension (8) :: timing

   print *, ' how many data items are there?'
   read *, how_many
```

```
   print *, ' what is the file  name?'
   read '(a)', file_name
   call date_and_time(values=timing)
   print *, ' initial'
   print *, timing(6), timing(7), timing(8)
   allocate (raw_data(how_many))
   call date_and_time(values=timing)
   print *, ' allocate'
   print *, timing(6), timing(7), timing(8)
   call read_data(file_name, raw_data,how_many)
   call date_and_time(values=timing)
   print *, ' read'
   print *, timing(6), timing(7), timing(8)
   call sort_data(raw_data,how_many)
   call date_and_time(values=timing)
   print *, ' sort'
   print *, timing(6), timing(7), timing(8)
   call print_data(raw_data,how_many)
   call date_and_time(values=timing)
   print *, ' print'
   print *, timing(6), timing(7), timing(8)
   print *, ' '
   print *, ' data written to file sorted.txt'
end program ch2005
```

20.6.1 Note – Recursive Subroutine

The actual sorting is done in the recursive subroutine `QuickSort`. The actual algorithm is taken from the Wirth book. See the bibliography for a reference.

Recursion provides us with a very clean and expressive way of solving many problems. There will be instances where it is worthwhile removing the overhead of recursion, but the first priority is the production of a program that is correct. It is pointless having a very efficient but incorrect solution.

We will look again at recursion and efficiency in a later chapter and see under what criteria we can replace recursion with iteration.

20.6.2 Note – Flexible Design

The QuickSort recursive routine can be replaced with another sorting algorithm and we can maintain the interface to Sort_data. We can thus decouple the implementation of the actual sorting routine from the defined interface. We would only need to recompile the Sort_data routine and we could relink using the already compiled main, read data and print data routines.

A later chapter looks at a non recursive implementation of quicksort where we look at some of the ways of rewriting the above program by replacing the recursive quicksort with the non recursive version.

20.6.3 Note – Timing Information

We call the date_and_time intrinsic subroutine to get timing information.

As can be seen it is the I/O that dominates the overall running time of the program. In the 10 years since first running this program we have seen the data set size increase from tens of thousands to tens and hundreds of millions.

20.7 Elemental Subroutines

We saw an example in Chap. 12 of elemental functions. Here is an example of an elemental subroutine.

```
module swap_module
implicit none
contains
    elemental subroutine swap(x,y)
    integer , intent(inout) :: x,y
    integer :: temp
        temp=x
        x=y
        y=temp
    end subroutine swap
end module swap_module

program ch2006
use swap_module
implicit none
integer , dimension(10) :: a,b
integer :: i
  do i=1,10
    a(i)=I
    b(i)=i*I
  end do
    print *,a
    print *,b
    call swap(a,b)
    print *,a
    print *,b
end program ch2006
```

The subroutine is written as if the arguments are scalar, but work with arrays! User defined elemental procedures came in with Fortran 95.

20.8 Summary

We now have a lot of the tools to start tackling problems in a structured and modular way, breaking problems down into manageable chunks and designing subprograms for each of the tasks.

20.9 Problems

1. Below is the random number program that was used to generate the data sets for the Quicksort example:

```
program ch2007
implicit none
integer :: n
integer :: i
real , allocatable , dimension (:) :: x
  print *,' how many  values ?'
  read *,n
  allocate(x(1:n))
  call random_number(x)
  x=x*1000
  open(unit=10,file='random.txt')
  do i=1,n
    write(10, 100)x(i)
    100 format(f8.3)
  end do
end program ch2007
```

   Run the Quick_Sort program in this chapter with the data file as input. Obtain timing details.

   What percentage of the time does the program spend in each subroutine? Is it worth trying to make the sort much more efficient given these timings?
2. Find out if there is a subroutine library like the NAG library available. if there is replace the Quick_Sort recursive subroutine with a suitable routine from that library. What times do you obtain?
3. Try using the operating system SORT command to sort the file. What timing figures do you get now?
   Was it worth writing a program?
4. Consider the following program:

```
program ch2008
!
! program to test array subscript checking
! when the array is passed as an argument.
!
implicit none
integer , parameter :: array_size=10
integer :: i
integer , dimension(array_size) :: a
  do i=1,array_size
    a(i)=I
  end do
  call sub01(a,array_size)
end program ch2008

subroutine sub01(a,array_size)
implicit none
integer , intent(in) :: array_size
integer , intent(in) , dimension(array_size) :: a
integer :: i
integer :: atotal=0
integer :: rtotal=0
  do i=1,array_size
    rtotal=rtotal+a(i)
  end do
  do i=1,array_size+1
    atotal=atotal+a(i)
  end do
  print *,' Apparent total is ' , atotal
  print *,'      real total is ' , rtotal
end subroutine sub01
```

The key thing to note is that we haven't used interface blocks and we have an error in the subroutine where we go outside the array. Run this program. What answer do you get for the apparent total?

Are there any compiler flags or switches which will enable you to trap this error?

5. Use the intrinsic functions `matmul` and `transfer` in program ch2003 to replace the current Fortran 77 style code.

20.10 Bibliography

Hoare, C.A.R.: Algorithm 63, partition; algorithm 64, quicksort, p.321; algorithm 65: FIND. Commun. ACM. **4**, 321–322 (1961)

Hoare, C.A.R.: Proof of a program: FIND. Commun. ACM. **13**(1), 39–45 (1970)
Hoare, C.A.R.: Proof of a recursive program: quicksort. Comput. J. **14**(4), 391–395 (1971)
Knuth, D.E.: The Art of Computer programming. Sorting and Searching, vol. 3. Addison-Wesley, Reading (1973)
Wirth, N.: Algorithms and Data Structures. Prentice-Hall, Upper Saddle River (1986)

20.11 Commercial Numerical and Statistical Subroutine Libraries

There are two major suppliers of commercial libraries:

- NAG: Numerical Algorithms Group

and

- Rogue Wave Software

They can be found at:

- http://www.nag.co.uk/

and

- http://www.roguewave.com/

respectively. Their libraries are written by numerical analysts, and are fully tested and well documented. They are under constant development and available for a wide range of hardware platforms and compilers. Parallel versions are also available.

Chapter 21
Modules

Common sense is the best distributed commodity in the world, for every man is convinced that he is well supplied with it.

Descartes

Aims

The aims of this chapter are to look at the facilities found in Fortran provided by modules, in particular:

- The use of a module to aid in the consistent definition of precision throughout a program and subprograms.
- The use of modules for global data.
- The use of modules for derived data types.
- Modules containing procedures
- Public, private and protected attributes
- The use statement and its extensions

21.1 Introduction

We have now covered the major executable building blocks in Fortran and they are

- The main program unit
- Functions
- Subroutines

and these provide us with the tools to solve many problems using just a main program and one or more external and internal procedures. Both external and internal procedures communicate through their argument lists, whilst internal procedures have access to data in their host program units.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_21, © Springer-Verlag London Limited 2012

We have also introduced modules. The first set of examples was in the chapter on functions. The second set were in the chapter on derived types and the third set were in the subroutine chapters.

We will now look at examples of modules for

- Precision definition.
- Global data
- Modules containing procedures
- Derived type definition.

Modules provide **the** code organisational mechanism in Fortran and can be thought of as the equivalent of classes in C++, Java and C#. They are one of the most important features of modern Fortran.

21.2 Basic Module Syntax

The form of a module is

```
module module_name
...
end module module_name
```

and the specifications and definitions contained within it is made available in the program units that need to access it by

```
use module_name
```

The `use` statement must be the first statement after the program, function or subroutine statement.

21.3 Modules for Global Data

So far the only way that a program unit can communicate with a procedure is through the argument list. Sometimes this is very cumbersome, especially if a number of procedures want access to the same data, and it means long argument lists. The problem can be solved using modules; e.g., by defining the precision to which you wish to work and any constants defined to that precision which may be needed by a number of procedures.

21.4 Modules for Precision Specification and Constant Definition

In the following example we use a module to define a parameter `long` to specify the precision to which we wish to work, and another for a range of mathematical constants including a value for the parameter π. Note that the parameter π is defined

to this working precision. We then import the module defining these parameters into the program units that need them. We also use a module procedure.

```
module precision_module
   implicit none
   integer , parameter :: &
long=selected_real_kind(15,307)
end module precision_module

module maths_module
   use precision_module
   implicit none
   real (long) , parameter :: c = 299792458.0_long
   ! units m s-1
   real (long) , parameter :: &
     e = 2.71828182845904523_long
   real (long) , parameter :: g = 9.812420_long
   ! 9.780 356 m s-2 at sea level on the equator
   ! 9.812 420 m s-2 at sea level in london
   ! 9.832 079 m s-2 at sea level at the poles
   real (long) , parameter :: &
     pi = 3.14159265358979323_long
end module maths_module

module sub1_module
implicit none
contains
subroutine sub1(radius,area,circum)
   use precision_module
   use maths_module
   implicit none
   real(long),intent(in)::radius
   real(long),intent(out)::area,circum
   area=pi*radius*radius
   circum=2.0_long*pi*radius
end subroutine sub1
end module sub1_module

program ch2101
   use precision_module
   use sub1_module
   implicit none
   real(long)::r,a,c
   integer ::I
   do i=1,5
     print*,'radius?'
```

```
    read*,r
    call sub1(r,a,c)
    print *,' for radius     = ',r
    print *,' area            = ',a
    print *,' circumference = ',c
  end do
end program ch2101
```

21.4.1 Note

In this example we wish to work with the precision specified by the kind type parameter `long` in the module `precision_module`. In order to do this we use the statement

```
use precision_module
```

inside the program units before any declarations. The kind type parameter `long` is then used with all the real type declaration e.g.,

```
real (long):: r ,a,c
```

To make sure that all floating point calculations are performed to the working precision specified by `long` any constants such as 2.0 in subroutine `Sub1` are specified as `const_long` e.g.,

```
2.0_long
```

Note also that we define things once and use them on two occasions, i.e., we define the precision once and use this definition in both the main program and the subroutine.

21.5 Modules for Sharing Arrays of Data

The following example uses a module to define a parameter and two arrays. The module also contains three subroutines that have access to the data in the module. The main program has the statement

```
use data_module
```

which interfaces to the three subroutines.

Note that in this example the calls to the subroutines have no parameters. They work with the data contained in the module.

```
module data_module
  implicit none
  integer , parameter    :: n=12
```

```
   real , dimension(1:n) :: rainfall
   real , dimension(1:n) :: sorted

contains
subroutine readdata
implicit none
integer :: i
character (len=40) :: filename
   print *,' What is the filename ?'
   read *,filename
   open(unit=100,file=filename)
   do i=1,n
     read (100,*) rainfall(i)
   end do
end subroutine readdata

subroutine sortdata
implicit none
   sorted=rainfall
   call selection

   contains

     subroutine selection
     implicit none
     integer :: i,j,k
     real :: minimum
       do i=1,n-1
          k=I
          minimum=sorted(i)
          do j=i+1,n
             if (sorted(j) < minimum) then
                k=j
                minimum=sorted(k)
             end if
          end do
          sorted(k)=sorted(i)
          sorted(i)=minimum
       end do
     end subroutine selection

end subroutine sortdata

subroutine printdata
implicit none
integer :: i
```

```
  print *,' original data is '
  do i=1,n
    print 100,rainfall(i)
    100 format(1x, f7.1)
  end do
  print *,' Sorted data is '
  do i=1,n
    print 100,sorted(i)
  end do
end subroutine printdata
end module data_module

program ch2102
use data_module
implicit none

  call readdata
  call sortdata
  call printdata

end program ch2102
```

21.6 Modules for Derived Data Types

When using derived data types and passing them as arguments to procedures, both the actual arguments and dummy arguments must be of the same type, i.e., they must be declared with reference to the same type definition. The only way this can be achieved is by using modules. The user defined type is declared in a module and each program unit that requires that type uses the module.

21.6.1 Person Data Type

In this example we have a user defined type `person` which we wish to use in the main program and pass arguments of this type to the subroutines `read_data` and `stats`. In order to have the type `person` available to two subroutines and the main program we have defined `person` in a module `personal_module` and then made the module available to each program unit with the statement

```
use personal_module
```

Note that we have put both subroutines in one module.

```
module personal_module
  implicit none
  type person
    real:: weight
```

```
    integer :: age
    character :: sex
  end type person
end module personal_module

module subs_module
use personal_module
implicit none
contains
subroutine read_data(data,max_no,no)
  implicit none
  type (person), dimension (:), intent(out)::data
  integer, intent(out):: no
  integer, intent(in):: max_no
  integer :: i
  do
    print *,'input number of patients'
    read *,no
    if (no > 0 .and. no <= max_no) exit
  end do
  do i=1,no
    print *,'for person',i
    print *,'weight ?'
    read*,data(i)%weight
    print*,'age ?'
    read*,data(i)%age
    print*,'sex ?'
    read*,data(i)%sex
  end do
end subroutine read_data

subroutine stats(data,no,m_a,f_a)
  implicit none
  type(person), dimension(:), intent(in) ::data
  real, intent(out) :: m_a,f_a
  integer, intent(in):: no
  integer :: i,no_f,no_m
  m_a=0.0; f_a=0.0;no_f=0; no_m =0
  do i=1,no
    if ( data(i)%sex == 'M' &
    .or. data(i)%sex == 'm') then
      m_a=m_a+data(i)%weight
      no_m=no_m+1
    elseif(data(i)%sex == 'F' &
      .or. data(i)%sex == 'f') then
      f_a=f_a+data(i)%weight
      no_f=no_f+1
    endif
```

```
   end do
   if (no_m > 0) then
     m_a = m_a/no_m
   endif
   if (no_f > 0 ) then
     f_a = f_a/no_f
   endif
end subroutine stats
end module subs_module

program ch2103
   use personal_module
   use subs_module
   implicit none
   integer ,parameter:: max_no=100
   type (person), dimension(1:max_no) :: patient
   integer :: no_of_patients
   real :: male_average, female_average
!
   call read_data(patient,max_no,no_of_patients)
   call stats(patient , no_of_patients ,  &
               male_average , female_average)
   print*, 'average male weight is    ',male_average
   print*, 'average female weight is  ',female_average
end program ch2103
```

21.7 Private, Public and Protected Attributes

With the examples of modules so far every entity in a module has been accessible to each program unit that 'uses' the module. By default all entities in a module have the public attribute, but sometimes it is desirable to limit the access. If entities have the private attribute this limits the possibility of inadvertent changes to a variable by another program unit.

Example of using public and private attributes:

```
real, public      :: a, b, c
integer, private  :: I, j, k
```

If a variable in a module is declared to be public, its access can be partially restricted by also giving it the protected attribute. This means that the variable can still be seen by program units that use the module but its value cannot be changed e.g.

```
integer, public, protected:: I
```

21.8 The Use Statement

In its simplest form the use statement is

```
use module_name
```

which then makes all the module's public entities available to the program unit. There may be times when only certain entities should be available to a particular program unit. In Example 1 `subroutine sub1` 'uses' `maths_module` but only needs `pi` and not `c`, `e` and `g`. The use statement could therefore be

```
use maths_module, only: pi
```

There are also times when an entity in a module needs to have its name changed when used in a program unit. For example variable `g` in `maths_module` needs to be called `gravity` in subroutine `sub1` so the `use` statement becomes

```
use maths_module, gravity=> g
```

21.9 Notes on Module Usage and Compilation

If we only have one file comprising all of the program units (main program, modules, functions and subroutines) then there is little to worry about. However, it is recommended that larger-scale programs be developed as a collection of files with related program units in each file, or even one program unit per file. This is more productive in the longer term, but it will lead to problems with modules unless we compile each module before we use it in other program units.

21.10 Formal Syntax

The following is taken from the standard and describes more fully requirements in the interface area.

21.10.1 Interface

The interface of a procedure determines the forms of reference through which it may be invoked. The procedure's interface consists of its name, binding label, generic identifiers, characteristics, and the names of its dummy arguments. The characteristics and binding label of a procedure are fixed, but the remainder of the interface may differ in differing contexts.

21.10.2 Implicit and Explicit Interfaces

Within the scope of a procedure identifier, the interface of the procedure is either explicit or implicit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit in such a scope.

The interface of a subroutine or a function with a separate result name is explicit within the subprogram where the name is accessible.

21.10.3 Explicit Interface

A procedure other than a statement function shall have an explicit interface if it is referenced and

- a reference to the procedure appears
  - with an argument keyword, or
  - in a context that requires it to be pure,
- the procedure has a dummy argument that
  - has the ALLOCATABLE, OPTIONAL, POINTER, TARGET, VALUE attribute,
  - is an assumed-shape array,
  - is a coarray,
  - is polymorphic,
- the procedure has a result that
  - is an array,
  - is a pointer or is allocatable, or
  - has a nonassumed type parameter value that is not a constant expression,
- the procedure is elemental

21.11 Summary

We have now introduced the concept of a module, another type of program unit, probably one of the most important features of Fortran 90. We have seen in this chapter how they can be used:

- Define global data.
- Define derived data types.
- Contain explicit procedure interfaces.
- Package together procedures.

This is a very powerful addition to the language, especially when constructing large programs and procedure libraries.

21.12 Problems

1. Write two functions, one to calculate the volume of a cylinder $\pi r^2 \, l$ where the radius is r and the length is l, and the other to calculate the area of the base of the cylinder $\pi \, r^2$. Define π as a parameter in a module which is used by the two functions. Now write a main program which prompts the user for the values of r and l, calls the two functions and prints out the results.
2. Make all the real variables in the above problem have 15 significant digits and a range of 10^{-307} to 10^{+307} . Use a module.

Chapter 22
Simple Data Structuring in Fortran

The good teacher is a guide who helps others to dispense with his services.

R. S. Peters, Ethics and Education

Aims

The aims of this chapter are to look at several complete examples illustrating data structuring in Fortran.

- Singly linked list: reading in an arbitrary amount of text
- Singly linked list: reading in an arbitrary quantity of numeric data
- Ragged arrays – lower triangular matrix
- Ragged arrays – variable sized data sets
- Perfectly balanced tree
- Date derived type

22.1 Introduction

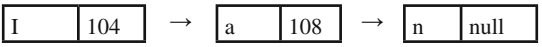
This chapter looks at simple data structuring in Fortran using a range of examples. We use modules throughout to define the data structures that we will be working with. The chapter starts with a number of pointer examples.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_22, © Springer-Verlag London Limited 2012

22.2 Singly Linked List: Reading in an Arbitrary Amount of Text

Conceptually a singly linked list consists of a sequence of boxes with compartments. In the simplest case the first compartment holds a data item and the second contains directions to the next box.

In the diagram below we have a singly linked list that holds three characters I, a and n. Element 1 is at address 100 and holds the letter I and a pointer to the next element – at address 104. Element 2 holds the letter a and a pointer to the next element – at address 108. Element 3 holds the letter n, and does not point to anything – we use the null pointer.



We can construct a data structure in Fortran to work with a singly linked list by defining a link data type with two components, a character and a pointer to a link data type. A complete program to do this is given below:

```
module link_module
  type link
    character :: c
    type (link) , pointer :: next => null()
  end type link
end module link_module

program ch2201
use link_module
implicit none
type (link) , pointer :: root , current
integer :: io_stat_number=0
  allocate(root)
  print *,' type in some text'
  read (unit = *, fmt = '(a)' , advance = 'no' , &
    iostat = io_stat_number) root%c
  if (io_stat_number == -1) then
    nullify(root%next)
  else
    allocate(root%next)
  endif
  current => root
  do while (associated(current%next))
    current => current%next
    read (unit=*,fmt='(a)',advance='no', &
```

```
    iostat=io_stat_number) current%c
    if (io_stat_number == -1) then
      nullify(current%next)
    else
      allocate(current%next)
    endif
  end do
  current => root
  do while (associated(current%next))
    print * , current%c
    current => current%next
  end do
end program ch2201
```

The behaviour of this program is system specific. You will have to look at your compiler documentation regarding the IO_Stat_Number. The first thing of interest is the type definition for the singly linked list. We have

```
module link_module
  type link
    character :: c
    type (link) , pointer    :: next => null()
  end type link
end module link_module
```

and we call the new type `link`. It comprises two component parts: the first holds a character C, and the second holds a pointer called `next` to allow us to refer to another instance of type `link`. Remember we are interested in joining together several boxes or `links`.

The next item of interest is the variable definition. Here we define two variables `root` and `current` to be pointers that point to items of type `link`. In Fortran when we define a variable to be a pointer we also have to define what it is allowed to point to. This is a very useful restriction on pointers, and helps make using them more secure.

The first executable statement

```
allocate(root)
```

requests that the variable `root` be allocated memory. At this time the contents of the character component is undefined and the pointer component is disassociated.

The next statement reads a character from the keyboard. We are using a number of additional features of the read statement, including

```
advance='no'
iostat=io_stat_number
```

and the two options combine to provide the ability to read an arbitrary amount of text from the user per line, and terminate only when end of file is encountered as the only input on a line, typically by typing CTRL Z. Note that the numbers returned by the `iostat` option are implementation specific. A small program would have to be written to test the values returned for each platform.

If an end of file is reached then the pointer `root%next` is nullified using the nullify statement. This gives the pointer a status of disassociated, and this is a convenient way of saying that it doesn't point to anything valid.

If the end of file is not detected then the next link in the chain is created.
The statement

```
Current => Root
```

means that both `Current` and `Root` point to the same physical memory location, and this holds a character data item and a pointer. We must do this as we have to know where the start of the list is. This is now our responsibility, not the compilers. Without this statement we are not able to do anything with the list except fill it up — hardly very useful.

The while loop is then repeated until end of file is reached. If the user had typed an end of file immediately then `Current%Next` would not be `associated`, and the while loop would be skipped.

This loop allocates memory and moves down the chain of boxes one character at a time filling in the links between the boxes as we go. We then have

```
Current => Root
```

and this now means that we are back at the start of the list, and in a position to traverse the list and print out each character in the list.

There is thus the concept with the pointer variable `current` of it providing us with a window into memory where the complete linked list is held, and we look at one part of the list at a time.

Both while loops use the intrinsic function `associated` to check the association status of a pointer.

It is recommended that this program be typed in, compiled and executed. It is surprisingly difficult to believe that it will actually read in a completely arbitrary number of characters from the user. Seeing is believing.

22.3 Singly Linked List: Reading in an Arbitrary Quantity of Numeric Data

In this example we will look at using a singly linked list to read in an arbitrary quantity of data and then allocating an array to copy it to for normal numeric calculations at run time:

```
module link_module
  type link
```

```
     real :: n
     type (link) , pointer :: next
   end type link
end module link_module

program ch2202_1
use link_module
implicit none
type (link) , pointer :: root, current
integer :: i=0
integer :: error=0
integer :: io_stat_number=0
integer :: blank_lines=0
real , allocatable , dimension (:) :: x
   allocate(root)
   print *,' Type in some numbers'
   read (unit = *, fmt = *, iostat = io_stat_number) &
   root%n
   if (io_stat_number > 0) then
     error=error+1
   else if (io_stat_number == -1) then
     nullify(root%next)
   else if (io_stat_number == -2) then
     blank_lines=blank_lines+1
   else
     i=i+1
     allocate(root%next)
  endif
  current => root
  do while (associated(current%next))
    current => current%next
    read (unit=*,fmt=*, iostat = io_stat_number) &
    current%n
    if (io_stat_number > 0) then
      error=error+1
    else if (io_stat_number == -1) then
      nullify(current%next)
    else if (io_stat_number == -2) then
      blank_lines=blank_lines+1
    else
      i=i+1
      allocate(current%next)
    endif
  end do
  print *,i,' items read'
```

```
  print *,blank_lines,' blank lines'
  print *,error,' items in error'
  allocate(x(1:i))
  I=1
  current => root
  do while (associated(current%next))
    x(i)=current%n
    i=I+1
    print * , current%n
    current => current%next
  end do
  print *,x
end program ch2202_1
```

Below is a variant on this using the NAG compiler. Note the use of a module (f90_iostat) and meaningful names for the status of the read:

```
module link_module
type link
   real :: n
   type (link) , pointer :: next
end type link
end module link_module

program ch2202_2
use link_module
use f90_iostat

type (link) , pointer :: root, current
integer :: i=0
integer :: io_stat_number=0
  allocate(root)
  print *,' Type in some numbers'
  read (unit = *, fmt = *, iostat = io_stat_number) &
    root%n
  if (io_stat_number == ioerr_eof) then
    nullify(root%next)
  else if(io_stat_number == ioerr_ok) then
    i=i+1
    allocate(root%next)
  endif
  current => root
  do while (associated(current%next))
    current => current%next
```

```
    read (unit=*,fmt=*, iostat=io_stat_number) &
      current%n
    if (io_stat_number == ioerr_eof) then
      nullify(current%next)
    else if(io_stat_number == ioerr_ok) then
      i=I+1
      allocate(current%next)
    endif
  end do
  print *,i,' items read'
  current => root
  do while (associated(current%next))
    print * , current%n
    current => current%next
  end do
end program ch2202_2
```

22.4 Ragged Arrays

Arrays in Fortran are rectangular, even when allocatable. However if you wish to set up a lower triangular matrix that uses minimal memory Fortran provides a number of ways of doing this. The following example achieves it using allocatable components.

```
module ragged_module
implicit none
type ragged
  real , dimension(:) , allocatable :: ragged_row
end type ragged
end module ragged_module

program ch2203
use ragged_module
implicit none
integer :: i
integer , parameter :: n=3
type (ragged) , dimension(1:n) :: lower_diag
  do i=1,n
    allocate(lower_diag(i)%ragged_row(1:i))
    print *,' type in the values for row ' , i
    read *,lower_diag(i)%ragged_row(1:i)
  end do
```

```
   do i=1,n
     print *,lower_diag(i)%ragged_row(1:i)
   end do
end program ch2203
```

Within the first do loop we allocate a row at a time and each time we go around the loop the array allocated increases in size.

22.5 Ragged Arrays and Variable Sized Data Sets

The previous example showed how to use allocatable components in a derived type to achieve ragged arrays. We extend this simple idea in the example below. In this example both the number of stations and the number of data items for each station is read in at run time and allocated accordingly. Notice that 0 is valid as the number of data items for a station.

```
module ragged_module
   type ragged
     real, allocatable, dimension (:) :: rainfall
   end type ragged
end module ragged_module

program ch2204
  use ragged_module
  implicit none
  integer :: i
  integer :: nr
  integer, allocatable, dimension (:) :: nc
  type (ragged), allocatable, dimension (:) :: station

  print *, ' enter number of stations'
  read *, nr
  allocate (station(1:nr))
  allocate (nc(1:nr))
  do i = 1, nr
     print *, ' enter the number of values ' , &
     ' for station ',i
     read *, nc(i)
     allocate (station(i)%rainfall(1:nc(i)))
     if (nc(i)==0) then
       cycle
     end if
     print *,' Type in the values for station ', i
```

```
    read *, station(i) %rainfall(1:nc(i))
  end do
  do i = 1, nr
    print *, ' Row ', i, ' Data = ', station(i)
%rainfall(1:nc(i))
  end do
end program ch2204
```

22.6 Perfectly Balanced Tree

Let us now look at a more complex example that builds a perfectly balanced tree and prints it out. A loose definition of a perfectly balanced tree is one that has minimum depth for n nodes. More accurately a tree is perfectly balanced if for each node the number of nodes in its left and right subtrees differ by at most 1:

```
module tree_node_module
  implicit none
  type tree_node
    integer :: number
    type (tree_node), pointer :: left, right
  end type tree_node
end module tree_node_module


module tree_module
  implicit none

contains
  recursive function tree(n) result (answer)
    use tree_node_module
    implicit none
    integer, intent (in) :: n
    type (tree_node), pointer :: answer
    type (tree_node), pointer :: new_node
    integer :: l, r, x

    if (n==0) then
      print *, ' terminate tree'
      nullify (answer)
    else
      l = n/2
      r = n - l - 1
      print *, l, r, n
      print *, ' next item'
      read *, x
```

```
      allocate (new_node)
      new_node%number = x
      print *, ' left branch'
      new_node%left => tree(l)
      print *, ' right branch'
      new_node%right => tree(r)
      answer => new_node
    end if
    print *, ' function tree ends'
  end function tree
end module tree_module

module print_tree_module
  implicit none

contains
  recursive subroutine print_tree(t,h)
    use tree_node_module
    implicit none
    type (tree_node), pointer :: t
    integer :: i
    integer :: h

    if (associated(t)) then
      call print_tree(t%left,h+1)
      do i = 1, h
        write (unit=*,fmt=10,advance='no')
        10 format ('    ')
      end do
      print *, t%number
      call print_tree(t%right,h+1)
    end if
  end subroutine print_tree
end module print_tree_module

program ch2205
! construction of a perfectly balanced tree
  use tree_node_module
  use tree_module
  use print_tree_module
  implicit none
  type (tree_node), pointer :: root
  integer :: n_of_items

  print *, 'enter number of items'
  read *, n_of_items
```

```
  root => tree(n_of_items)

  call print_tree(root,0)
end program ch2205
```

There are a number of very important concepts contained in this example and they include:

- The use of a module to define a type. For user defined data types we must create a module to define the data type if we want it to be available in more than one program unit.
- The use of a function that returns a pointer as a result.
- As the function returns a pointer we must determine the allocation status before the function terminates. This means that in the above case we use the `nullify(result)` statement. The other option is to target the pointer.
- The use of `associated` to determine if the node of the tree is terminated or points to another node.

Type the program in and compile, link and run it. Note that the tree only has the minimal depth necessary to store all of the items. Experiment with the number of items and watch the tree change its depth to match the number of items.

22.7 Date Class

The following is a complete manual rewrite of Skip Noble and Alan Millers date module. The original worked with the built-in Fortran intrinsic data types. It has been rewritten to work with a derived date data type.

The first key code segment is

```
  type, public :: date
    private
    integer :: day
    integer :: month
    integer :: year
  end type date
```

where the `date` data type is public but its components are private. This means that access to the components must be done via subroutines and functions within the `date_module` module.

The next key code segment is

```
  public :: calendar_to_julian, &
    date_, &
    date_stamp, &
    date_to_day_in_year, &
```

```
    date_to_weekday_number, &
    get_day, &
    get_month, &
    get_year, &
    julian_to_date, &
    julian_to_date_and_week_and_day, &
    ndays, &
    year_and_day_to_date
```

where we explicitly make the listed subroutines and functions public, as the code segment from the top of the module,

```
! ..
! .. Default Accessibility ..
  private
```

defines everything to be private.

We have to provide a user defined constructor when the components of the derived type are private. This is given below:

```
  function date_(dd,mm,yyyy) result (x)
! .. implicit none Statement ..
    implicit none
! ..
! .. function return value ..
    type (date) :: x
! ..
! .. Scalar Arguments ..
    integer, intent (in) :: dd, mm, yyyy
! ..
    x = date(dd,mm,yyyy)
  end function date_
```

This in turn calls the built-in constructor date. As the `date_` function is now an executable statement we cannot initialise in a declaration, i.e. the following is not allowed.

```
  type (date) :: date1_(11,2,1952)
```

We also provide three additional procedures to access the components of the date class:

```
  get_day
  get_month
  get_year
```

This is common programming practice in object oriented and object based programming.

The program has also been through the Nag tool suite and this has helped to systematically lay out the code.

```
module date_module
  ! Collected and put together January 1972,
  ! h. d. knoble.
  ! Original references are cited in each
  ! routine.
  ! Code converted using to_f90 by alan
  ! miller
  ! Date: 1999-12-22 time: 10:23:47
  ! Compatible with imagine1 f compiler:
  ! 2002-07-19
  ! At this time the functions and
  ! subroutines were as described below
  ! FUNCTION iday(yyyy, mm, dd) RESULT(ival)
  ! FUNCTION izlr(yyyy, mm, dd) RESULT(ival)
  ! SUBROUTINE calend(yyyy, ddd, mm, dd)
  ! SUBROUTINE cdate(jd, yyyy, mm, dd)
  ! SUBROUTINE daysub(jd, yyyy, mm, dd, wd,
  ! ddd)
  ! FUNCTION jd(yyyy, mm, dd) RESULT(ival)
  ! FUNCTION ndays(mm1, dd1, yyyy1,
  ! mm2, dd2, yyyy2) RESULT(ival)
  ! SUBROUTINE date_stamp( string, want_day,
  ! short )
  ! Code converted by ian chivers and jane
  ! sleightholme
  ! November 2004 - May 2005
  ! The changes are to go from
  ! working with integer variables
  ! for year, day and month to
  ! user defined date variables.
  ! .. Implicit None Statement ..
  implicit none
  ! ..
  ! .. Default Accessibility ..
  private
  ! ..
  ! .. Derived Type Declarations ..
  type, public :: date
    private
    integer :: day
    integer :: month
```

```
    integer :: year
  end type date
  ! ..
  ! .. Public Statements ..
  public :: calendar_to_julian, date_, &
    date_stamp, date_to_day_in_year, &
    date_to_weekday_number, get_day, &
    get_month, get_year, julian_to_date, &
    julian_to_date_and_week_and_day, ndays, &
    year_and_day_to_date
  ! ..
  ! The above are the contained
  ! functions and subroutines
  ! in this module.
  ! Here is a short description of each one
  ! date_to_day_in_year - function
  ! returns the day in the year
  ! original arguments of day,month,year
  ! now date
  ! dayinyear
  ! date_to_weekday_number - function
  ! returns the week day number
  ! original argument d,m,y
  ! now date
  ! weekdaynum
  ! year_and_day_to_date - subroutine
  ! returns the day and month from
  ! year and day in year
  ! julian_to_date - subroutine
  ! returns a year_and_day_to_datear date
  ! from
  ! a julian date
  ! ndays - function
  ! returns the number of days between
  ! two dates
  ! julian_to_date_and_week_and_day -
  ! subroutine
  ! given a julian day this routine
  ! calculates year, month day and
  ! week day number and day number
  ! calendar_to_julian - function
  ! returns julian date from
  ! year_and_day_to_datear date
contains
  ! arithmetic functions "izlr" and "iday"
  ! are taken from remark on
```

```
! algorithm 398, by j. douglas robertson,
! cacm 15(10):918.
function date_to_day_in_year(x)
   ! Convert from date to day in year
   ! .. Implicit None Statement ..
   implicit none
   ! ..
   ! .. Function Return Value ..
   integer :: date_to_day_in_year
   ! ..
   ! .. Structure Arguments ..
   type (date), intent (in) :: x
   ! ..
   ! .. Intrinsic Functions ..
   intrinsic modulo
   ! ..
   date_to_day_in_year = 3055*(x%month+2)/ &
      100 - (x%month+10)/13*2 - 91 + &
      (1-(modulo(x%year,4)+3)/4+(modulo(x% &
      year,100)+99)/100-(modulo(x%year, &
      400)+399)/400)*(x%month+10)/13 + x%day
end function date_to_day_in_year

function date_to_weekday_number(x)
   ! .. Implicit None Statement ..
   implicit none
   ! ..
   ! .. Function Return Value ..
   integer :: date_to_weekday_number
   ! ..
   ! .. Structure Arguments ..
   type (date), intent (in) :: x
   ! ..
   ! .. Intrinsic Functions ..
   intrinsic modulo
   ! ..
   date_to_weekday_number = modulo((13*( &
     x%month+10-(x%month+10)/13*12)-1)/5+x% &
     day+77+5*(x%year+(x%month- &
     14)/12-(x%year+(x%month-14)/12)/100*100 &
     )/4+(x%year+(x%month-14)/12)/400-(x% &
     year+(x%month-14)/12)/100*2,7)
end function date_to_weekday_number

function year_and_day_to_date(year,day) &
    result (x)
```

```
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    type (date) :: x
    ! ..
    ! .. Scalar Arguments ..
    integer, intent (in) :: day, year
    ! ..
    ! .. Local Scalars ..
    integer :: t
    ! ..
    ! .. Intrinsic Functions ..
    intrinsic modulo
    ! ..
    x%year = year
    t = 0
    if (modulo(year,4)==0) then
      t = 1
    end if
    ! ------the following statement is
    ! necessary IF year is < 1900 or > 2100.
    if (modulo(year,400)/=0 .and. &
        modulo(year,100)==0) then
      t = 0
    end if
    x%day = day
    if (day>59+t) then
      x%day = x%day + 2 - t
    end if
    x%month = ((x%day+91)*100)/3055
    x%day = (x%day+91) - (x%month*3055)/100
    x%month = x%month - 2
    if (x%month>=1 .and. x%month<=12) then
      return
    end if
    ! x%month will be correct
    ! iff day is correct for year.
    write (unit=*,fmt='(a,i11,a)') '$$year_and&
      &_day_to_date: day of the year &
      &input =', day, ' is out of range.'
  end function year_and_day_to_date

  function julian_to_date(julian) result (x)
    ! Given a julian day number the date is
```

```
    ! returned.
    ! julian is the julian date from an epoch
    ! in the very distant past. see cacm 1968
    ! 11(10):657,
    ! letter to the editor by fliegel and van
    ! flandern.
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Scalar Arguments ..
    integer, intent (in) :: julian
    ! ..
    ! .. Local Scalars ..
    integer :: l, n
    ! ..
    ! .. Function Return Value ..
    type (date) :: x
    ! ..
    l = julian + 68569
    n = 4*l/146097
    l = l - (146097*n+3)/4
    x%year = 4000*(l+1)/1461001
    l = l - 1461*x%year/4 + 31
    x%month = 80*l/2447
    x%day = l - 2447*x%month/80
    l = x%month/11
    x%month = x%month + 2 - 12*l
    x%year = 100*(n-49) + x%year + 1
  end function julian_to_date

  subroutine julian_to_date_and_week_and_day &
       (jd,x,wd,ddd)
    ! given jd, a julian day # (see asf jd),
    ! this routine calculates dd,
    ! the day number of the month;
    ! mm, the month number; yyyy the year;
    ! wd the weekday number, and
    ! ddd the day number of the year.
    ! example:
    ! CALL julian_to_date_and_week_and_day
    ! (2440588, yyyy, mm, dd, wd, ddd)
    ! yields 1970 1 1 4 1.
    ! .. Implicit None Statement ..
    implicit none
    ! ..
```

```
    ! .. Scalar Arguments ..
    integer, intent (out) :: ddd, wd
    integer, intent (in) :: jd
    ! ..
    ! .. Structure Arguments ..
    type (date), intent (out) :: x
    ! ..
    x = julian_to_date(jd)
    wd = date_to_weekday_number(x)
    ddd = date_to_day_in_year(x)
  end subroutine &
    julian_to_date_and_week_and_day

  function calendar_to_julian(x) &
       result (ival)
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    integer :: ival
    ! ..
    ! .. Structure Arguments ..
    type (date), intent (in) :: x
    ! ..
    ! date routine calendar_to_julian
    ! converts date to
    ! julian date. see cacm 1968 11(10):657,
    ! letter to the
    ! editor by henry f. fliegel and
    ! thomas c. van flandern.
    ! example calendar_to_julian(1970, 1, 1)
    ! = 2440588
    ival = x%day - 32075 + 1461*(x%year+4800+ &
      (x%month-14)/12)/4 + 367*(x%month-2-((x &
      %month-14)/12)*12)/12 - 3*((x%year+4900 &
      +(x%month-14)/12)/100)/4
  end function calendar_to_julian

  function ndays(date1,date2)
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    integer :: ndays
    ! ..
    ! .. Structure Arguments ..
```

```
    type (date), intent (in) :: date1, date2
    ! ..
    ! dates; that is mm1/dd1/yyyy1 minus
    ! mm2/dd2/yyyy2,
    ! where datei and datej have elements mm,
    ! dd, yyyy.
    ! ndays will be positive iff
    ! date1 is more recent than date2.
    ndays = calendar_to_julian(date1) - &
      calendar_to_julian(date2)
  end function ndays

  subroutine date_stamp(string,want_day, &
      short)
    ! Returns the current date as a character
    ! string
    ! e.g.
    ! want_day short string
    ! .TRUE. .TRUE. Thursday, 23 Dec 1999
    ! .TRUE. .FALSE. Thursday, 23 December
    ! 1999
    ! <- default/
    ! .FALSE. .TRUE. 23 Dec 1999
    ! .FALSE. .FALSE. 23 December 1999
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Scalar Arguments ..
    logical, optional, intent (in) :: short, &
      want_day
    character (*), intent (out) :: string
    ! ..
    ! .. Local Scalars ..
    integer :: pos
    logical :: sh, want_d
    ! ..
    ! .. Local Arrays ..
    integer :: val(8)
    character (9) :: day(0:6) = (/ &
      'Sunday    ','Monday   ',  'Tuesday  ', &
      'Wednesday', 'Thursday ', 'Friday   ', &
      'Saturday '/)
    character (9) :: month(1:12) &
      = (/ 'January   ', 'February ', &
      'March     ', 'April     ', 'May     ', &
      'June      ', 'July      ', 'August  ', &
```

```
      'September', 'October  ', 'November ', &
      'December '/)
    ! ..
    ! .. Intrinsic Functions ..
    intrinsic date_and_time, len_trim, &
      present, trim
    ! ..
    ! .. Local Structures ..
    type (date) :: x
    ! ..
    want_d = .true.
    if (present(want_day)) want_d = want_day
    sh = .false.
    if (present(short)) sh = short
    call date_and_time(values=val)
    x = date_(val(3),val(2),val(1))
    if (want_d) then
      pos = date_to_weekday_number(x)
      string = trim(day(pos)) // ','
      pos = len_trim(string) + 2
    else
      pos = 1
      string = ' '
    end if
    write (string(pos:pos+1),'(i2)') val(3)
    if (sh) then
      string(pos+3:pos+5) = month(val(2)) &
        (1:3)
      pos = pos + 7
    else
      string(pos+3:) = month(val(2))
      pos = len_trim(string) + 2
    end if
    write (string(pos:pos+3),'(i4)') val(1)
    return
  end subroutine date_stamp

  function date_(dd,mm,yyyy) result (x)
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    type (date) :: x
    ! ..
    ! .. Scalar Arguments ..
```

```
    integer, intent (in) :: dd, mm, yyyy
    ! ..
    x = date(dd,mm,yyyy)
  end function date_

  function get_year(x)
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    integer :: get_year
    ! ..
    ! .. Structure Arguments ..
    type (date), intent (in) :: x
    ! ..
    get_year = x%year
  end function get_year

  function get_month(x)
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    integer :: get_month
    ! ..
    ! .. Structure Arguments ..
    type (date), intent (in) :: x
    ! ..
    get_month = x%month
  end function get_month

  function get_day(x)
    ! .. Implicit None Statement ..
    implicit none
    ! ..
    ! .. Function Return Value ..
    integer :: get_day
    ! ..
    ! .. Structure Arguments ..
    type (date), intent (in) :: x
    ! ..
    get_day = x%day
  end function get_day
end module date_module
```

```
program ch2206
  ! .. Use Statements ..
  use date_module, only : calendar_to_julian, &
    date, date_, date_stamp, &
    date_to_day_in_year, &
    date_to_weekday_number, get_day, &
    get_month, get_year, &
    julian_to_date_and_week_and_day, ndays, &
    year_and_day_to_date
  ! ..
  ! .. Implicit None Statement ..
  implicit none
  ! ..
  ! .. Local Scalars ..
  integer :: dd, ddd, i, mm, ndiff, wd, yyyy
  character (50) :: message
  ! ..
  ! .. Local Arrays ..
  integer :: val(8)
  ! ..
  ! .. Intrinsic Functions ..
  ! compute date this year for changing
  ! clocks
  ! back to est.
  ! i.e. compute date for the last
  ! Sunday in October for this year.
  intrinsic date_and_time
  ! ..
  ! .. Local Structures ..
  type (date) :: date1, date2, x
  ! ..
  ! The following no longer works as the data
  ! components of the type are private
  ! type (date) :: birthday=date_(11,2,1952)
  ! Test date_stamp
  message = ' date_stamp = '
  call date_stamp(message(15:))
  write (*,'(a)') message
  message = ' date_stamp = '
  call date_stamp(message(15:), &
     want_day=.false.)
  write (*,'(a)') message
  message = ' date_stamp = '
  call date_stamp(message(15:),short=.true.)
  write (*,'(a)') message
```

```
message = ' date_stamp = '
call date_stamp(message(15:), &
  want_day=.false.,short=.true.)
write (*,'(a)') message
call date_and_time(values=val)
yyyy = val(1)
mm = 10
do i = 31, 26, -1
   x = date_(i,mm,yyyy)
   if (date_to_weekday_number(x)==0) then
     print *, 'turn clocks back to est on: '
     print *, i, ' October ', get_year(x)
     exit
   end if
end do
! compute date this year for
! turning clocks ahead to dst
! i.e., compute date for the first
! Sunday in April for this year.
call date_and_time(values=val)
yyyy = val(1)
mm = 4
do i = 1, 8
   x = date_(i,mm,yyyy)
   if (date_to_weekday_number(x)==0) then
     print *, &
       'turn clocks ahead to dst on: '
     print *, i, ' April ', get_year(x)
     exit
   end if
end do
call date_and_time(values=val)
yyyy = val(1)
mm = 12
dd = 31
x = date_(dd,mm,yyyy)
! is this a leap year? i.e., is
! 12/31/yyyy the 366th day of the year?
if (date_to_day_in_year(x)==366) then
   print *, get_year(x), ' is a leap year'
else
  print *, get_year(x), &
     ' is not a leap year'
end if
x = date_(1,1,1970)
```

```
  call julian_to_date_and_week_and_day &
     (calendar_to_julian(x),x,wd,ddd)
  if (get_year(x)/=1970 .or. get_month(x)/=1 &
       .or. get_day(x)/=1 .or. wd/=4 .or. &
       ddd/=1) then
     print *, 'julian_to_date_and_week_and_day &
       &failed'
     print *, ' date, wd, ddd = ', &
       get_year(x), get_month(x), get_day(x), &
       wd, ddd
     stop
  end if
  ! difference between to same
  ! months and days over 1 leap year is 366.
  date1 = date_(22,5,1984)
  date2 = date_(22,5,1983)
  ndiff = ndays(date1,date2)
  yyyy = 1970
  x = year_and_day_to_date(yyyy,ddd)
  if (ndiff/=366) then
     print *, 'ndays failed; ndiff = ', ndiff
  else
     ! recover month and day
     ! from year and day number.
     if (get_month(x)/=1 .and. get_day(x)/=1) &
         then
       print *, 'year_and_day_to_date failed'
       print *, ' mma, dda = ', get_month(x), &
         get_day(x)
     else
       print *, &
         '** date manipulation subroutines'
       print *, '** simple test ok.'
     end if
  end if
end program ch2206
```

There are wrap problems with some of the complex arithmetic expressions. The version on the web site is obviously correct.

We also have an alternate form of array declaration in this program, which is given below. It is common in Fortran 77 style code:

```
  integer :: val(8)
```

The next major addition to this code would be a date checking routine to test the validity of dates. This would be called from within our constructor date_. This

would mean that we could never have an invalid date when using the date_module. This is left as a programming exercise.

22.7.1 Notes: DST in the USA

The above program is no longer correct. Beginning in 2007, Daylight Saving Time was brought forward by 3 or 4 weeks in Spring and extended by 1 week in the Fall. Daylight Saving Time begins for most of the United States at 2 a.m. on the second Sunday of March. Time reverts to standard time at 2 a.m. on the first Sunday in November.

22.8 Problems

1. Compile and run the examples in this chapter with your compiler. Do they all work with your compiler? You may have problems with the examples that use allocatable components. Not all compilers support this feature at this time.
2. Modify the ragged array example that processes a lower triangular matrix to work with an upper triangular matrix.
3. Using the balanced tree example as a basis and modify it to work with a character array rather than an integer. The routine that prints the tree will also have to be modified to reflect this.
4. Modify the Date program to account for the current DST in the USA.

22.9 Bibliography

Schneider, G.M., Bruell, S.C.: Advanced Programming and Problem Solving with Pascal. Wiley, New York (1981)

The book is aimed at computer science students and follows the curriculum guidelines laid down in Communications of the ACM, August 1985, Course CS2. The book is very good for the complete beginner as the examples are very clearly laid out and well explained. There is a coverage of data structures, abstract data types and their implementation, algorithms for sorting and searching, the principles of software development as they relate to the specification, design, implementation and verification of programs in an orderly and disciplined fashion – their words.

Vowels, R.A.: Algorithms and Data Structures in F and Fortran. Unicomp, Tucson (1998)

The only book currently that uses Fortran 90/95 and F. Visit the Fortran web site for more details. They are the publishers.

http://www.fortran.com/fortran/market.html

Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs (1976)

An early but illuminating book on the subject. Well worth a read. Pascal is used.

Wirth, N.: Algorithms + Data Structures. Prentice-Hall, Englewood Cliffs (1986)

This is the Modula 2 version. Closer to Fortran than the Pascal version.

Chapter 23
Operator Overloading

All the persons in this book are real and none is fictitious even in part.

Flann O'Brien, The Hard Life

Aims

The aims of this chapter are to look at operator overloading in Fortran.

23.1 Introduction

In programming operator overloading can be regarded as a way of achieving polymorphism in that operators (e.g. +,−,*,/ or =) can have different implementations depending on the types of their arguments.

In some programming languages overloading is defined by the language. In Fortran for example, the addition + operator invokes quite different code when used with integer, real or complex types.

Some languages allow the programmer to implement support for user defined types. Fortran introduced support for operator and assignment overloading in the 1990 standard.

23.2 Other Languages

Operator overloading is not new and several languages offer support for the feature including:

- Algol 68 – 1968
- Ada – Ada 83

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_23, © Springer-Verlag London Limited 2012

- C++ – First standard, 1998
- Eiffel – 1986
- C# – 2001

Java, however does not.

23.3 Example

The following example overloads the addition operator.

```
module T_Position
implicit none
type Position
   integer :: X
   integer :: Y
   integer :: Z
end type Position

interface operator (+)
   module procedure New_Position
end interface

contains

function New_Position(A,B)
type (Position) ,intent(in) :: A,B
type (Position) :: New_Position
   New_Position % X = A % X + B % X
   New_Position % Y = A % Y + B % Y
   New_Position % Z = A % Z + B % Z
end function New_Position

end module T_Position

program ch2301
use T_Position
implicit none
type (Position) :: A,B,C
   A%X=10
   A%Y=10
   A%Z=10
   B%X=20
   B%Y=20
   B%Z=20
```

```
   C=A+B
   print *,A
   print *,B
   print *,C
end program ch2301
```

We have extended the meaning of the addition operator so that we can write simple expressions in Fortran based on it and have our new position calculated using a user supplied function that actually implements the calculation of the new position.

23.4 Problem

1. Compile and run this example. Overload the subtraction operator as well.

Chapter 24
Generic Programming

General notions are generally wrong.

Letter to Mr. Wortley Montegu, 28th March 1710.

Aims

This chapter looks at an example that implements generic programming in Fortran.

24.1 Introduction

Fortran 77 had several generic functions, e.g. the sine function could be called with arguments of type real, double precision or complex. Fortran 90 extended the idea so that a programmer could write their own generic functions or subroutines. For example we can now write a sort routine works with arguments of a variety of types, e.g. integer, real etc.

24.2 Generic Programming and Other Languages

Generic programming has a wider meaning in computer science and effectively is a style of computer programming in which an algorithm is written once, but can be made to work with a variety of types.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_24, © Springer-Verlag London Limited 2012

This style of programming is provided in several programming languages and in a variety of ways.
Languages that support generics include

- Ada
- C#
- Eiffel
- Java

C++ supports the functionality via templates.

To quote the generic programming pioneer Alexander Stepanov;

- Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.

and quoting Bjarne Stroustrup:

- lift algorithms and data structures from concrete examples to their most general and abstract form.

It is called parametric polymorphism in the languages ML and Haskell.

The term parameterised type is used in the book Design Patterns: Elements of Reusable Object-Oriented Software. The authors, sometimes called the Gang of Four, or GoF, also state that

- dynamic, highly parameterized software is harder to understand and build than more static software.

Ada was one of the first languages to support generic programming, and the paper

- David R. Musser and Alexander A. Stepanov: A library of generic algorithms in Ada. Proceedings of the 1987 Annual ACM SIGAda international conference on Ada, pages 216–225.

shows how old the ideas are.

We'll look at a concrete example in Fortran next.

24.3 Generic Example

Simplistically, a procedure is generic if it can handle arguments of more than one data type. The example we will use is based on the earlier one of sorting. In the original example the program worked with real data. In the example below we have extended the program to handle both integer and real data.

What is not obvious from our use of the internal procedures is that there will be specific procedures to handle each data type, i.e., if a function can take integer, real

and complex arguments then there will be one implementation of that function for each data type, i.e., three separate functions.

In the example below we add the ability to handle integer data. This means that where we had:

- read data
- sort data
- print data

and one subroutine to implement the above we now have two subroutines to do each of the above, one to handle integers and one to handle reals:

```
program ch2401

use read_data_module
use sort_data_module
use print_data_module

implicit none
integer :: How_Many
character (len=20) :: File_Name
integer , allocatable , dimension(:) :: integer_data
real    , allocatable , dimension(:) :: real_data

  print * , ' How many data items are there?'
  read  * , How_Many
  print * , ' What is the file name?'
  read '(A)',File_Name
  allocate(integer_data(How_Many))
  call read_data(File_Name,integer_data,How_Many)
  call Sort_data(integer_data,How_Many)
  call print_data(integer_data,How_Many)
  print *, ' Phase 1 ends.'
  print *, ' data written to file name isorted.txt'
  deallocate(integer_data)

  print * , ' How many data items are there?'
  read  * , How_Many
  print * , ' What is the file name?'
  read '(A)',File_Name
  allocate(real_data(How_Many))
  call read_data(File_Name,real_data,How_Many)
  call Sort_data(real_data,How_Many)
  call print_data(real_data,How_Many)
  print *, ' program ends.'
  print *, ' data written to file name rsorted.txt'
end program ch2401
```

```
module read_data_module

interface read_data

  module procedure read_integer
  module procedure read_real

end interface read_data

contains

subroutine read_real(File_Name,Raw_data,How_Many)
implicit none
character (len=*) , intent(in) :: File_Name
integer , intent(in) :: How_Many
real , intent(out) , &
   dimension(:) :: Raw_data
integer :: i
   open(file=File_Name,unit=1)
   do i=1,How_Many
     read (unit=1,fmt=*) Raw_data(i)
   enddo
end subroutine read_real

subroutine read_integer(File_Name,Raw_data,How_Many)
implicit none
character (len=*) , intent(in):: File_Name
integer , intent(in) :: How_Many
integer , intent(out) , &
   dimension(:) :: Raw_data
integer :: i
  open(file=File_Name,unit=1)
  do i=1,How_Many
    read (unit=1,fmt=*) Raw_data(i)
  enddo
end subroutine read_integer

end module read_data_module

module sort_data_module

interface sort_data

  module procedure sort_integer
  module procedure sort_real

end interface sort_data

contains
```

```
subroutine sort_real(Raw_data,How_Many)
implicit none
integer , intent(in) :: How_Many
real , intent(inout) , &
  dimension(:) :: Raw_data
  call QuickSort(1,How_Many)

contains

recursive subroutine QuickSort(L,R)
implicit none
integer , intent(in) :: L,R
integer :: i,j
real :: V,T

  i=l
  j=r
  v=raw_data( int((l+r)/2) )
  do
    do while (raw_data(i) < v )
         i=I+1
    enddo
    do while (v < raw_data(j) )
         j=j-1
    enddo
    if (i<=j) then
      t=raw_data(i)
      raw_data(i)=raw_data(j)
      raw_data(j)=t
      i=I+1
      j=j-1
    endif
    if (i>j) exit
  enddo

  if (l<j) then
    call quicksort(l,j)
  endif

  if (i<r) then
    call quicksort(i,r)
  endif

end subroutine QuickSort

end subroutine sort_real

subroutine sort_integer(Raw_data,How_Many)
```

```
implicit none
integer , intent(in) :: How_Many
integer , intent(inout) , &
  dimension(:) :: Raw_data
  call QuickSort(1,How_Many)

contains

recursive subroutine QuickSort(L,R)
implicit none
integer , intent(in) :: L,R
integer :: i,j
integer :: V,T

  i=l
  j=r
  v=raw_data( int((l+r)/2) )
  do
    do while (raw_data(i) < v )
        i=I+1
    enddo
    do while (v < raw_data(j) )
        j=j-1
    enddo
    if (i<=j) then
      t=raw_data(i)
      raw_data(i)=raw_data(j)
      raw_data(j)=t
      i=I+1
      j=j-1
    endif
    if (i>j) exit
  enddo

  if (l<j) then
    call quicksort(l,j)
  endif

  if (i<r) then
    call quicksort(i,r)
  endif

end subroutine QuickSort

end subroutine sort_integer

end module sort_data_module

module print_data_module
```

```
interface print_data

  module procedure print_integer
  module procedure print_real

end interface print_data

contains

subroutine print_real(Raw_data,How_Many)
implicit none
integer , intent(in) :: How_Many
real , intent(in) , &
   dimension(:) :: Raw_data
integer :: i
  open(file='rsorted.txt',unit=2)
  do i=1,How_Many
    write(unit=2,fmt=*) Raw_data(i)
  end do
  close(2)
end subroutine print_real

subroutine print_integer(Raw_data,How_Many)
implicit none
integer , intent(in) :: How_Many
integer , intent(in) , &
   dimension(:) :: Raw_data
integer :: i
  open(file='isorted.txt',unit=2)
  do i=1,How_Many
    write(unit=2,fmt=*) Raw_data(i)
  end do
  close(2)
end subroutine print_integer

end module print_data_module
```

The key code is given below for each module:

```
interface read_data

  module procedure read_integer
  module procedure read_real

end interface read_data

interface sort_data

  module procedure sort_integer
  module procedure sort_real
```

```
end interface sort_data

interface print_data

  module procedure print_integer
  module procedure print_real

end interface print_data
```

The interface block name is used in the calling routine and the appropriate module procedure will be called, based on a signature match of the actual and dummy parameters.

This is quite useful, but not as useful as the functionality provided in other languages. Have a look at the following two examples which show the code for a generic quicksort in C++ and C#.

24.3.1 Generic Quicksort in C++

```
template <class Type>
void swap(Type array[],int i, int j)
{
  Type tmp=array[i];
  array[i]=array[j];
  array[j]=tmp;
}
template <class Type>
void quicksort( Type array[], int l, int r)
{
  int i=l;
  int j=r;
  Type v=array[int((l+r)/2)];
  for (;;)
  {
     while (array[i] < v) i=i+1;
     while (v < array[j]) j =j−1;
     if (i<=j) { swap(array, i,j); i=i+1 ; j=j-1; }
     if (i>j) goto ended ;
  }
ended: ;
   if (l<j) quicksort(array,l,j);
   if (i<r) quicksort(array,i,r);
}
template <class Type>
void print(Type array[],int size)
{
```

```
  cout << " [ " ;
  for (int ix=0;ix<size; ++ix)
     cout << array[ix] << " ";
  cout << "] \n";
}
#include <iostream>
using namespace std;
int main()
{
  double da[]={1.9,8.2,3.7,6.4,5.5,1.8,9.2,3.6,7.4,5.5};
  int ia[]={1,10,2,9,3,8,4,7,6,5};
  int size=sizeof(da)/sizeof(double);
  cout << " Quicksort of double array is \n";
  quicksort(da,0,size-1);
  print(da,size);
  size=sizeof(ia)/sizeof(int);
  cout << " Quicksort of integer array is \n";
  quicksort(ia,0,size-1);
  print(ia,size);
  return(0);
}
```

24.3.2 Generic Quicksort in C#

```
using System;

public static class generic02
{
  public static void swap< Type > (Type[] array,int I,
int j)
  {
    Type tmp=array[i];
    array[i]=array[j];
    array[j]=tmp;
  }

  public static void quicksort< Type >( Type[] array,
int l, int r)
    where Type : IComparable< Type >
  {
    int i=l;
    int j=r;
    Type v=array[(int)((l+r)/2)];
    for (;;)     {
  while (array[i].CompareTo( v) < 0 ) i=I+1;
```

```
        while (v.CompareTo(array[j]) < 0) j=j-1;
        if (i<=j) { swap(array,i,j); i=I+1 ; j=j-1; }
        if (i>j) goto ended ;
    }
    ended: ;
    if (l<j) quicksort(array,l,j);
    if (i<r) quicksort(array,i,r);
  }
  public static void print< Type > (Type[] array,int
size)
  {
    int I;
    int l;
    l=array.Length;
    for (i=0;i<l;i++)
      Console.WriteLine(array[i]);
  }

  public static int Main()
  {
    double[]
da={1.9,8.2,3.7,6.4,5.5,1.8,9.2,3.6,7.4,5.5};
     int[]    ia={1,10,2,9,3,8,4,7,6,5};
     int size;

     size=da.Length;
     Console.WriteLine("Original array");
     print(da,size);
     quicksort(da,0,size-1);
     Console.WriteLine("Sorted array");
     print(da,size);

     size=ia.Length;
     Console.WriteLine("Original array");

     print(ia,size);
     quicksort(ia,0,size-1);
     Console.WriteLine("Sorted array");
     print(ia,size);
     return(0);
  }
}
```

24.3.3 Summary

Note in both the C++ and C# case we only have one version of the algorithm. Fortran still has a way to go! Maybe Fortran 2020?

24.4 Problem

1. Write a generic swap routine, that swaps two rank 1 integer arrays and two rank 1 real arrays.

24.5 Bibliography

This site is a collection of Alex Stepanov's papers, class notes, and source code, covering generic programming and other topics.

http://www.stepanovpapers.com/

C++
C++ Templates: The Complete Guide, David Vandevoorde, Nicolai M Josuttis, 2003 Addison-Wesley. ISBN 0–201–73484–2

C#
Visit the following site
http://msdn.microsoft.com/en-us/library/512aeb7t(v=vs.80).aspx
for a very good coverage of generics and C#.

Chapter 25
Mathematical Examples

You look at science (or at least talk of it) as some sort of demoralising invention of man, something apart from real life, and which must be cautiously guarded and kept separate from everyday existence. But science and everyday life cannot and should not be separated. Science, for me, gives a partial explanation for life. In so far as it goes, it is based on fact, experience and experiment.

Rosalind Franklin.

Aims

The aims of this chapter are to look at several mathematical examples in Fortran.

- Using linked lists for sparse matrix problems.
- The solution of a set of ordinary differential equations using the Runge–Kutta–Merson method, with the use of a procedure as a parameter, and the use of work arrays.
- Diagonal extraction of a matrix.
- The solution of linear equations using Gaussian elimination
- An elemental e**x function

25.1 Introduction

This chapter looks at a small number of mathematical examples in Fortran.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_25, © Springer-Verlag London Limited 2012

25.2 Using Linked Lists for Sparse Matrix Problems

A matrix is said to be sparse if many of its elements are zero. Mathematical models in areas such as management science, power systems analysis, circuit theory and structural analysis consist of very large sparse systems of linear equations. It is not possible to solve these systems with classical methods because the sparsity would be lost and the eventual system would become too large to solve. Many of these systems consist of tens of thousands, hundreds of thousands and millions of equations. As computer systems become ever more powerful with massive amounts of memory the solution of even larger problems becomes feasible.

Direct Methods for Sparse Matrices, by Duff I.S., Erismon A.M. and Reid J.K., looks at direct methods for solving sparse systems of linear equations.

Sparse matrix techniques lend themselves to the use of dynamic data structures in Fortran. Only the nonzero elements of a sparse matrix need be stored, together with their positions in the matrix. Other information also needs to be stored so that row or column manipulation can be performed without repeated scanning of a potentially very large data structure. Sparse methods may involve introducing some new nonzero elements, and a way is needed of inserting them into the data structure. This is where the Fortran pointer construct can be used. The sparse matrix can be implemented using a linked list to which entries can be easily added and from which they can be easily deleted.

As a simple introduction, consider the storage of sparse vectors. What we learn here can easily be applied to sparse matrices, which can be thought of as sets of sparse vectors.

25.2.1 Inner Product of Two Sparse Vectors

Assume that we have two sparse vectors x and y, for example:

$$\underline{x} = \begin{bmatrix} 3 \\ 0 \\ 5 \\ 0 \\ 0 \\ 4 \end{bmatrix} \qquad \underline{y} = \begin{bmatrix} 0 \\ 1 \\ 3 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

and we wish to calculate the inner product $x^T y \equiv \sum_{i=1}^{n} x_i \quad y_i$. There are a number of approaches to doing this and the one we use in the program below stores

them as two linked lists. Only the nonzero elements are stored (together with their indices):

```
    x data file      y data file
       3  1            1  2
       5  3            3  3
       4  6            2  5
          1               6
```

Here is the program.

```
program ch2501

! this program reads the non-zero elements of
! two sparse vectors x and y together with their
! indices, and stores them in two linked lists.
! using these linked lists it then calculates
! and prints out the inner product.
! it also prints the values.

! updated 21/3/00 to initialise pointers to
! be disassociated using intrinsic function null
! plus minor updates

implicit none
character (len=30) :: filename
type sparse_vector
  integer :: index
  real :: value
  type (sparse_vector), pointer :: next => null()
end type sparse_vector
type (sparse_vector), pointer :: &
  root_x, current_x,root_y, current_y
real :: inner_prod = 0.0
integer :: io_status

! read non-zero elements of vector x together
! with indices into a linked list
  print *, 'input file name for vector x'
  read '(a)', filename
  open (unit=1, file=filename , &
    status='old',iostat=io_status)
  if (io_status/=0) then
    print *, 'error opening file ', filename
    stop
  end if
  allocate (root_x)
  read (unit=1,fmt=*,iostat=io_status) &
    root_x%value, root_x%index
```

```
  if (io_status/=0) then
    print *, ' error reading from file ' , &
      filename, ' or file empty'
    stop
  end if

! read data for vector x from file until eof

  current_x => root_x
  allocate (current_x%next)
  do while (associated(current_x%next))
    current_x => current_x%next
    read (unit=1,fmt=*,iostat=io_status) &
      current_x%value, current_x%index
    if (io_status==0) then
      allocate (current_x%next)
      cycle
    else if (io_status>0) then

! error on reading

      print *, 'error occurred when reading from ' , &
        filename
      stop
    else

! end of file

      nullify (current_x%next)
    end if
  end do
  close (unit=1)

! read non-zero elements of vector y together
! with indices into a linked list

  print *, 'input file name for vector y'
  read '(a)' , filename
  open (unit=1, file=filename, &
    status='old',iostat=io_status)
  if (io_status/=0) then
    print *, 'error opening file ', filename
    stop
  end if
  allocate (root_y)
  read (unit=1,fmt=*,iostat=io_status) &
    root_y%value,root_y%index
```

```
  if (io_status/=0) then
    print *, ' error when reading from ', filename, &
      'or file empty'
    stop
  end if

! read data for vector y from file until eof

  current_y => root_y
  allocate (current_y%next)
  do while (associated(current_y%next))
    current_y => current_y%next
    read (unit=1,fmt=*,iostat=io_status) &
      current_y%value, current_y%index
    if (io_status==0) then
      allocate (current_y%next)
      cycle
    else if (io_status>0) then

! error on reading

      print *, 'error occurred when reading from ', &
        filename
      stop
    else
! end of file

      nullify (current_y%next)
    end if
  end do

! data has now been read and stored in 2 linked lists
! start at the beginning of x linked list and
! y linked list and compare indices
! in order to perform inner product

  current_x => root_x
  current_y => root_y
  do while (associated(current_x%next))
    do while &
     ( associated(current_y%next) .and. &
       current_y%index<current_x%index)

! move through 2nd list

      current_y => current_y%next
    end do
```

```
! at this point current_y%index >= current_x%index
! or 2nd list is exhausted

    if (current_y%index==current_x%index) then
      inner_prod = inner_prod + &
      current_x%value*current_y%value
    end if
    current_x => current_x%next
  end do

! print out inner product

  print *, 'inner product of two sparse vectors is :',&
    inner_prod

! print non-zero values of vector x and indices

  print *, 'non-zero values of vector x and indices:'
  current_x => root_x
  do while (associated(current_x%next))
    print *, current_x%value, current_x%index
    current_x => current_x%next
  end do

! print non-zero values of vector y and indices

  print *, 'non-zero values of vector y and indices:'
  current_y => root_y
  do while (associated(current_y%next))
    print *, current_y%value, current_y%index
    current_y => current_y%next
  end do

end program ch2501
```

25.3 Solving a System of First-Order Ordinary Differential Equations Using Runge–Kutta–Merson

Simulation and mathematical modelling of a wide range of physical processes often leads to a system of ordinary differential equations to be solved. Such equations also occur when approximate techniques are applied to more complex problems. We will restrict ourselves to a class of ordinary differential equations called initial value problems. These are systems for which all conditions are given at the same value of the independent variable. We will further restrict ourselves to first-order initial value problems of the form:

$$\frac{dy_1}{dt} = f_1(\underline{y}, t)$$
$$\frac{dy_2}{dt} = f_2(\underline{y}, t)$$
$$\cdots$$
$$\frac{dy_n}{dt} = f_n(\underline{y}, t)$$

or

$$\dot{\underline{y}} = \underline{f}(\underline{y}, t) \tag{25.1}$$

with initial conditions

$$\underline{y}(t0) = \underline{y}0$$

where

$$\underline{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \qquad \underline{f} = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} \qquad \underline{y_0} = \begin{pmatrix} y_1(t_0) \\ \vdots \\ y_n(t_0) \end{pmatrix}$$

if we have a system of ordinary differential equations of higher order then they can be reformulated to a system of order one. See the NAG library documentation for solving ordinary differential equations.

One well-known class of methods for solving initial value ordinary differential equations is Runge–Kutta. In this example we have coded the Runge–Kutta–Merson algorithm, which is a fourth-order method and solves (25.1) from a point t = A to a point t = B.

It starts with a step length $h = (B - A)/100$ and includes a local error control strategy such that the solution at t + h is accepted if:

$$|error\ estimate| < user\ defined\ tolerance$$

If this isn't satisfied the step length h is halved and the solution attempt is repeated until the above is satisfied or the step length is too small and the problem is left unsolved. If the error criterion is satisfied the algorithm progresses with a suitable step length solving the equations at intermediate points until the end point B is reached. For a full discussion of the algorithm and the error control mechanism used see Numerical Methods in Practice by Tim Hopkins and Chris Phillips:

```
module precision_module
integer,parameter:: long=selected_real_kind(15,307)
end module precision_module
```

```
module rkm_module
use precision_module
implicit none
contains
subroutine runge_kutta_merson(y,fun,ifail,n,a,b,tol)
!
! runge-kutta-merson method for the solution
! of a system of n
! 1st order initial value ordinary
! differential equations.
! the routine tries to integrate from t=a to t=b with
! initial conditions in y, subject to the
! condition that the
! absolute error estimate <= tol. the step length is
! adjusted automatically to meet this condition.
! if the routine is successful it returns with
! ifail = 0, t=b and
! the solution in y.
!
!
implicit none
! define arguments
!
real (long),intent(inout):: y(:)
real(long), intent(in)::a,b,tol
integer,intent(in)::n
integer,intent(out)::ifail
!
interface
  subroutine fun(t,y,f,n)
  use precision_module
  implicit none
  real(long),intent(in),dimension(:)::y
  real(long),intent(out),dimension(:)::f
  real(long),intent(in)::t
  integer,intent(in)::n
  end subroutine fun
end interface
!
! local variables
!
real(long), dimension(1:size(y)):: &
          s1,s2,s3,s4,s5,new_y_1,new_y_2,error
real(long)::t,h,h2,h3,h6,h8,factor=1.e-2_long
real(long)::smallest_step=1.e-6_long,max_error
```

```
integer::no_of_steps=0
!
  ifail=0
!
! check input parameters
!
  if(n <= 0 .or. a == b .or. tol <= 0.0) then
    ifail = 1
    return
  endif
!
! initialize t to be start of interval and
! h to be 1/100 of interval
  t=a
  h=(b-a)/100.0_long
  do                    ! beginning of repeat loop
    h2=h/2.0_long
    h3=h/3.0_long
    h6=h/6.0_long
    h8=h/8.0_long
!
! calculate s1,s2,s3,s4,s5
!
! s1=f(t,y)

    call fun(t,y,s1,n)

    new_y_1=y+h3*s1

! s2 = f(t+h/3,y+h/3*s1)

    call fun(t+h3,new_y_1,s2,n)
    new_y_1=y+h6*s1+h6*s2

! s3=f(t+h/3,y+h/6*s1+h/6*s2)

    call fun(t+h3,new_y_1,s3,n)

    new_y_1=y+h8*(s2+3.0_long*s3)

! s4=f(t+h/2,y+h/8*(s2+3*s3))

    call fun(t+h2,new_y_1,s4,n)
    new_y_1=y+h2*(s1-3.0_long*s3+4.0_long*s4)

! s5=f(t+h,y+h/2*(s1-3*s3+4*s4))
```

```
    call fun(t+h,new_y_1,s5,n)
!
! calculate values at t+h
!
    new_y_1=y+h6*(s1+4.0_long*s4+s5)
    new_y_2=y+h2*(s1-3.0_long*s3+4.0_long*s4)
!
! calculate error estimate
!
    error=abs(0.2_long*(new_y_1-new_y_2))
    max_error=maxval(error)
    if(max_error > tol) then
!
! halve step length and try again
!
       if(abs(h2) < smallest_step) then
         ifail = 2
         return
       endif
       h=h2
    else
!
! accepted approximation so overwrite y with y_new_1,
! and t with t+h
!
       y=new_y_1
       t=t+h
!
! can next step be doubled?
!
       if(max_error*factor < tol)then
         h=h*2.0_long
       endif
!
! does next step go beyond interval end b,
! if so set h = b-t
!
       if(t+h > b) then
         h=b-t
       endif
       no_of_steps=no_of_steps+1
    endif
    if(t >= b) exit             ! end of repeat loop
  end do
end subroutine runge_kutta_merson
end module rkm_module
```

A main program to use this subroutine is of the form:

```
program ch2502
use precision_module
use rkm_module
use fun1_module
implicit none
real(long),dimension(:),allocatable::y
real(long)::a,b,tol
integer::n,ifail,all_stat
!
  print *,'input no of equations'
  read*,n
!
! allocate space for y - checking to see that it
! allocates properly
!
  allocate(y(1:n),stat=all_stat)
  if(all_stat /= 0) then
    print * , ' not enough memory'
    print * , ' array y is not allocated'
    stop
  endif
  print *,' input start and end of interval over'
  print *,' which equations to be solved'
  read *,a,b
  print *,"input initial conditions"
  read *,y(1:n)
  print *,'input tolerance'
  read *,tol
  print 100,a
  100 format('at t= ',f5.2,' initial conditions are :')
  print 200,y(1:n)
  200 format(4(f5.2,2x))
  call runge_kutta_merson(y,fun1,ifail,n,a,b,tol)
  if(ifail /= 0) then
    print *,'integration stopped with ifail = ',ifail
  else
    print 300,b
    300 format('at t= ',f5.2,' solution is:')
    print 200,y(1:n)
  endif
end program ch2502
```

Consider trying to solve the following system of first-order ordinary differential equations:

$$\dot{y}_1 = \tan y_3$$

$$\dot{y}_2 = \frac{-0.032 \tan y_3}{y_2} - \frac{0.02 y_2}{\cos y_3}$$

$$\dot{y}_3 = -\frac{0.032}{y_2^2}$$

over an interval $t = 0.0$ to $t = 8.0$ with initial conditions

$$y1 = 0 \quad y2 = 0.5 \quad y3 = \frac{\pi}{5}$$

The user supplied subroutine, packaged as a module procedure, is:

```
module fun1_module

implicit none
contains
subroutine fun1(t,y,f,n)
  use precision_module
  implicit none
  real(long),intent(in),dimension(:)::y
  real(long),intent(out),dimension(:)::f
  real(long),intent(in)::t
  integer,intent(in)::n
!
  f(1)=tan(y(3))
  f(2)=-0.032_long*f(1)/y(2)-0.02_long*y(2)/cos(y(3))
  f(3)=-0.032_long/(y(2)*y(2))
end subroutine fun1
end module fun1_module
```

25.3.1 Note: Alternative Form of the Allocate Statement

In the main program Odes we have defined Y to be a deferred-shape array, allocating it space after the variable N is read in. In order to make sure that enough memory is available to allocate space to array Y the allocate statement is used as follows:

allocate(Y(1:N),STAT = All_stat)

if the allocation is successful variable All_stat returns zero; otherwise it is given a processor dependent positive value. We have included code to check for this and the program stops if All_stat is not zero.

25.3.2 Note: Automatic Arrays

The subroutine `runge_kutta_merson` needs a number of local rank 1 arrays S1, S2, S3, S4 and S5 for workspace, their shape and size being the same as the dummy argument Y. Fortran supplies automatic arrays for this purpose and can be declared as

```
real(Long), dimension (1:SIZE(Y)) :: S1, S2, S3, S4, S5
```

The size of automatic arrays can depend on the size of actual arrays: in our example they are the same shape and size as the dummy array Y, or some other dummy arguments. Automatic arrays are created when the procedure is called and destroyed when control passes back to the calling program unit. They may have different shapes and sizes with different calls to the procedure, and because of this automatic arrays cannot be saved or initialised.

A word of warning should be given at this point. If there isn't enough memory available when an automatic array needs to be created problems will occur. Unlike allocatable arrays there is no way of testing to see if an automatic array has been created successfully. The general feeling is that even though they are nice, automatic arrays should be used with care and perhaps shouldn't be used in production code!

25.3.3 Note: Subroutine as a Dummy Procedure Argument

In order to make the use of subroutine `runge_kutta_merson` as general as possible the user can choose the name of the subroutine in which the actual system of equations to be solved is defined. In this case we have chosen `fun1` as the name of the subroutine, which is then used as an actual argument when calling `runge_kutta_merson` from the main program e.g.

```
call runge_kutta_merson(y,fun1,ifail,n,a,b,tol)
```

An explicit interface for subroutine `fun1` is provided by it being contained in a module.

The equivalent dummy subroutine argument is `fun` and this needs an explicit interface in the subroutine `runge_kutta_merson`.

25.3.4 Note: Compilation When Using Modules

When compiling this program and the modules they must be done in the correct order:

- precision_module
- rkm_module
- fun1_module

and then

- the main program.

25.4 A Subroutine to Extract the Diagonal Elements of a Matrix

A common task mathematically is to extract the diagonal elements of a matrix. For example if

$$A = \begin{pmatrix} 21 & 6 & 7 \\ 9 & 3 & 2 \\ 4 & 1 & 8 \end{pmatrix}$$

the diagonal elements are (21, 3, 8).

This can be thought of as extracting an array section, but the intrinsic function `pack` is needed. In its simplest form `pack (array,vector)` packs an array, Array, into a rank 1 array, Vector, according to Array's array element order.

Below is a complete program to demonstrate this:

```
module md_module
implicit none
contains
subroutine matrix_diagonal(a,diag,n)
implicit none
real, intent (in), dimension (:,:) :: a
real, intent (out), dimension (:) :: diag
integer, intent (in) :: n
real, dimension(1:size(a,1)*size(a,1)):: temp
!
! subroutine to extract the diagonal elements of an n
* n matrix A
!
   temp = pack(a, .true.)
   diag = temp(1:n*n:n+1)
end subroutine matrix_diagonal
end module md_module

program ch2503
! program reads the n * n matrix from a file
use md_module
implicit none
integer :: i, n
real, allocatable, dimension (:,:) :: a
real, allocatable, dimension (:) :: adiag
character (len=20) :: filename
  print *, 'input name of data file'
```

```
  read '(a)', filename
  open (unit=1,file=filename)
  read (1,*) n
  allocate (a(1:n,1:n),adiag(1:n))
  do i = 1, n
    read (1,*) a(i,1:n)
  end do
  call matrix_diagonal(a,adiag,n)
  print *, ' diagonal elements of a are:'
  print *, adiag
end program ch2503
```

25.5 The Solution of Linear Equations Using Gaussian Elimination

At this stage we have introduced many of the concepts needed to write numerical code, and have included a popular algorithm, Gaussian elimination, together with a main program which uses it and a module to bring together many of the features covered so far.

Finding the solution of a system of linear equations is very common in scientific and engineering problems, either as a direct physical problem or indirectly, for example, as the result of using finite difference methods to solve a partial differential equation. We will restrict ourselves to the case where the number of equations and the number of unknowns are the same. The problem can be defined as:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{12}x_2 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad &= \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{25.1}$$

or

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}$$

which can be written as:

$$A\,x = b \tag{25.2}$$

where A is the n x n coefficient matrix, b is the right-hand-side vector and x is the vector of unknowns. We will also restrict ourselves to the case where A is a general real matrix.

Note that there is a unique solution to (25.2) if the inverse, A^{-1}, of the coefficient matrix A, exists. However, the system should never be solved by finding A^{-1} and then solving $A^{-1}\ b = x$ because of the problems of rounding error and the computational costs.

A well-known method for solving (25.2) is Gaussian elimination, where multiples of equations are subtracted from others so that the coefficients below the diagonal become zero, producing a system of the form:

$$\begin{pmatrix} a_{11}^* & a_{12}^* & \dots & a_{1n}^* \\ 0 & a_{22}^* & \dots & a_{2n}^* \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & a_{nn}^* \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^* \\ b_2^* \\ \dots \\ b_n^* \end{pmatrix}$$

where A has been transformed into an upper triangular matrix. By a process of backward substitution the values of x drop out.

The subroutine `gaussian_elimination` implements the Gaussian elimination algorithm with partial pivoting, which ensure that the multipliers are less than 1 in magnitude, by interchanging rows if necessary. This is to try and prevent the buildup of errors.

This implementation is based on two LINPACK routines SGEFA and SGESL and a Fortran 77 subroutine written by Tim Hopkins and Chris Phillips and found in their book Numerical Methods in Practice.

The matrix A and vector B are passed to the subroutine `Gaussian_Elimination` and on exit both A and B are overwritten. Mathematically Gaussian elimination is described as working on rows, and using partial pivoting row interchanges may be necessary. Due to Fortran's row element ordering, to implement this algorithm efficiently it works on columns rather than rows by interchanging elements within a column if necessary.

```
module precision_module
implicit none
integer, parameter :: long = selected_real_kind(15,307)
end module precision_module

module ge_module
implicit none
contains
  subroutine gaussian_elimination(a,n,b,x,singular)
! routine to solve a system ax=b
```

```
! using gaussian elimination
! with partial pivoting
! the code is based on the linpack routines
! sgefa and sgesl
! and operates on columns rather than rows!
  use precision_module
  implicit none
! matrix a and vector b are over-written
! arguments
  integer, intent (in) :: n
  real (long), intent (inout) :: a(:,:), b(:)
  real (long), intent (out) :: x(:)
  logical, intent (out) :: singular
! local variables
  integer :: i, j, k, pivot_row
  real (long) :: pivot, sum, element
  real (long), parameter :: eps = 1.e-13_long

! work through the matrix column by column

    do k = 1, n - 1

! find largest element in column k for pivot

    pivot_row = maxval(maxloc(abs(a(k:n,k)))) + k -
1

! test to see if a is singular
! if so return to main program

      if (abs(a(pivot_row,k))<=eps) then
        singular = .true.
        return
      else
        singular = .false.
      end if

! exchange elements in column k if largest is
! not on the diagonal

       if (pivot_row/=k) then
          element = a(pivot_row,k)
          a(pivot_row,k) = a(k,k)
          a(k,k) = element
          element = b(pivot_row)
```

```
         b(pivot_row) = b(k)
         b(k) = element
       end if

! compute multipliers
! elements of column k below diagonal
! are set to these multipliers for use
! in elimination later on

       a(k+1:n,k) = a(k+1:n,k)/a(k,k)

! row elimination performed by columns for efficiency

       do j = k + 1, n
         pivot = a(pivot_row,j)
         if (pivot_row/=k) then
!          swap if pivot row is not k
            a(pivot_row,j) = a(k,j)
            a(k,j) = pivot
         end if
         a(k+1:n,j) = a(k+1:n,j) - pivot*a(k+1:n,k)
       end do

! apply same operations to b

       b(k+1:n) = b(k+1:n) - a(k+1:n,k)*b(k)
     end do

! backward substitution

     do i = n, 1, -1
       sum = 0.0
       do j = I + 1, n
         sum = sum + a(i,j)*x(j)
       end do
       x(i) = (b(i)-sum)/a(i,i)
     end do
   end subroutine gaussian_elimination
end module ge_module

program ch2504
use precision_module
use ge_module
implicit none
integer :: i, n
```

```
real (long), allocatable :: a(:,:), b(:), x(:)
logical :: singular

   print *, 'number of equations?'
   read *, n
   allocate (a(1:n,1:n),b(1:n),x(1:n))
   do i = 1, n
     print *, 'input elements of row ', i, ' of a'
     read *, a(i,1:n)
     print *, 'input element ', i, ' of b'
     read *, b(i)
   end do
   call gaussian_elimination(a,n,b,x,singular)
   if (singular) then
     print *, 'matrix is singular'
   else
     print *, 'solution x:'
     print *, x(1:n)
   end if
end program ch2504
```

25.5.1 Notes

25.5.1.1 Module for Kind Type

A module, `precision_module`, has been used to define a kind type parameter, `Long`, to specify the floating point precision to which we wish to work. This module is then used by the main program and the subroutine, and the kind type parameter `Long` is used with all the real type definitions and with any constants, e.g.,

```
real(Long), parameter :: Eps=1.E-13_Long
```

25.5.1.2 Deferred-Shape Arrays

In the main program matrix A and vectors B and X are declared as deferred-shape arrays, by specifying their rank only and using the allocatable attribute. Their shape is determined at run time when the variable N is read in and then the statement

```
allocate(A(1:N,1:N), B(1:N), X(1:N))
```

is used.

25.5.1.3 Intrinsic Functions `maxval` and `maxloc`

In the context of subroutine `gaussian_elimination` we have used:

```
maxval ( maxloc (abs ( a ( k:n,k ) ) ) ) + k - 1
```

Breaking this down,

```
maxloc ( abs ( a (k:n,k) ) )
```

takes the rank 1 array

$$(|A(K,K)|, |A(K+1,K)|, \ldots |A(N,K)|) \tag{25.3}$$

where $|A(K,K)|$ = ABS(A(K,K)) and of length $-K+1$. It returns the position of the largest element as a rank 1 array of size one, e.g., (L)

Applying `maxval` to this rank 1 array (L) returns L as a scalar, L being the position of the largest element of array (25.3).

What we actually want is the position of the largest element of (25.3), but in the K^{th} column of matrix A. We therefore have to add K-1 to L to give the actual position in column K of A.

25.6 Allocatable Function Results

A function may return an array, and in this example the array allocation takes place in the function.

```
module running_average_module
implicit none
contains
function running_average(r,how_many) result(rarray)
integer , intent(in) :: how_many
real , intent(in) , allocatable , dimension(:) :: r
real , allocatable , dimension(:) :: rarray
integer :: i
real :: sum=0.0
   allocate(rarray(1:how_many))
   do i=1,how_many
     sum = sum + r(i)
     rarray(i)=sum/I
   end do
end function running_average
end module running_average_module

module read_data_module
implicit none
```

```
contains
subroutine read_data(file_name,raw_data,how_many)
implicit none
character (len=*) , intent(in) :: file_name
integer , intent(in) :: how_many
real , intent(out) , allocatable , dimension(:) ::
raw_data
integer :: i
   allocate(raw_data(1:how_many))
   open(file=file_name,unit=1)
   do i=1,how_many
     read (unit=1,fmt=*) raw_data(i)
   enddo
end subroutine read_data
end module read_data_module
```

```
program ch2505
use running_average_module
use read_data_module
implicit none
integer :: how_many
character (len=20) :: file_name
real , allocatable , dimension(:)  :: raw_data
real , allocatable , dimension(:)  :: ra
integer :: i
   print * , ' how many data items are there?'
   read  * , how_many
   print * , ' what is the file name?'
   read '(a)',file_name
   call read_data(file_name,raw_data,how_many)
   allocate(ra(1:how_many))
   ra=running_average(raw_data,how_many)
   do i=1,how_many
     print *,raw_data(i),'    ' ,ra(i)
   end do
end program ch2505
```

This facility was introduced in Fortran 95.

25.7 Elemental e**x Function

The following is an elemental version of the etox function covered in an earlier chapter.

```
module etox_module
implicit none
contains
    elemental real function etox(x)
    implicit none
    real , intent(in) :: x
    real :: term
    integer :: nterm
    real , parameter :: tol =1.0e-6
      etox=1.0
      term=1.0
      nterm=0
      do
        nterm=nterm+1
        term=(x/nterm)*term
        etox=etox+term
        if (term<=tol) exit
      end do
    end function etox
end module etox_module
```

```
program ch2506
use etox_module
implicit none
integer :: i
real :: x
real , dimension(10) :: y
  x=1.0
  do i=1, 10
    y(i)=I
  end do
  print *,y
  x=etox(x)
  print *,x
  y=etox(y)
  print *,y
end program ch2506
```

Elemental functions require the use of explicit interfaces, and we have therefore used modules to achieve this.

25.8 Problems

1. Compile and run the sparse matrix example with the data provided.
2. Compile and run the Runge Kutta Merson example with the data provided.
3. Compile and run the Gaussian Elimination example with the following data.

$$A = \begin{pmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{pmatrix}$$

$$b = \begin{pmatrix} -359 \\ 281 \\ 85 \end{pmatrix}$$

and the solution is

$$x = \begin{pmatrix} 1 \\ -2 \\ -5 \end{pmatrix}$$

25.9 Bibliography

Duff, I.S., Erismon, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford Science Publications, Oxford (1986)

- Authoritative coverage of this area. Relatively old, but well regarded. Code segments and examples are a mixture of Fortran 77 and Algol 60 (which of course do not support pointers) and therefore the implementation of linked lists is done using the existing features of these languages. The onus is on the programmer to correctly implement linked lists using fixed size arrays rather than using the features provided by pointers in a language. It is remarkable how elegant these solutions are, given the lack of dynamic data structures in these two languages.

Hopkins, T., Phillips, C.: Numerical Methods in Practice, Using the NAG Library. Addison-Wesley, Wokingham/Reading (1988)

- Good adjunct to the NAG library documentation for the less numerate user.

Schneider, G.M., Bruell, S.C.: Advanced Programming and Problem Solving with Pascal. Wiley, New York (1981)

- The book is aimed at computer science students and follows the curriculum guidelines laid down in Communications of the ACM, August 1985, Course CS2. The book is very good for the complete beginner as the examples are very

clearly laid out and well explained. There is a coverage of data structures, abstract data types and their implementation, algorithms for sorting and searching, the principles of software development as they relate to the specification, design, implementation and verification of programs in an orderly and disciplined fashion — their words.

Vowels, R.A.: Algorithms and Data Structures in F and Fortran. Unicomp, Tucson (1998)

- The only book currently that uses Fortran 90/95 and F. Visit the Fortran web site for more details. They are the publishers.
- http://www.fortran.com/fortran/market.html

Wirth, N.: Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs (1976)

- An early but illuminating book on the subject. Well worth a read. Pascal is used.

Wirth, N.: Algorithms + Data Structures. Prentice-Hall, Englewood Cliffs (1986)

- This is the Modula 2 version. Closer to Fortran than the Pascal version.

Commercial numerical libraries
NAG. Visit their web site for up to date details of their products:

- http://www.nag.co.uk/

Rogue Wave Software. Visit their web site for details of their products:

- http://www.roguewave.com/

Chapter 26
Object Oriented Programming

"For Madmen only"

Hermann Hesse, Steppenwolf

Aims

The aims of this chapter are to look at object oriented programming in Fortran.

26.1 Introduction

This chapter looks at object oriented programming in Fortran. The chapter on programming languages covers the topic in a broader context.

26.2 Brief Review of the History of Object Oriented Programming

Object oriented programming is not new. One of the first languages to offer support was Simula 67, a language designed for discrete event simulation by Ole Johan Dahl, Bjorn Myhrhaug and Kristen Nygaard whilst working at the Norwegian Computing Centre in Oslo in the 1960s.

One of the next major developments was in the 1970s at the Xerox Palo Alto Research Centre Learning Research Group who began working on a vision of the ways different people might effectively use computing power. One of the outcomes of their work was the Smalltalk 80 system. Objects are at the core of the Smalltalk 80 system.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_26, © Springer-Verlag London Limited 2012

The 1980s and 1990s saw a number of object oriented programming languages emerge. They include

- Eiffel. Bertrand Meyer, Eiffel Software.
- C++ from C with classes. Bjarne Stroustrup at Bell Labs.
- Oberon 2. Niklaus Wirth at ETH in Zurich.
- Java. James Gosling, originally Sun, now Oracle.

C# is a recent Microsoft addition to the list.

Object-oriented programming is effectively a programming methodology or paradigm using objects (data structures made up of data and methods). We will use the concept of a shape class in our explanation and examples. The Simula Begin book starts with shapes, and it is often used in introductions to object oriented programming in other languages.

Some of the key concepts are

- encapsulation or information hiding – the implementation of the data is hidden inside an object and clients or users of the data only have access to an abstract view of it. Methods are used to access and manipulate the data. For example a shape class may have an x and y position, and methods exist to get and set the positions and draw and move the shape.
- data abstraction – if we have an abstract shape data type we can create multiple variables of that type.
- inheritance – an existing abstract data type can be extended. It will inherit the data and methods from the base type and add additional data and methods. A key to inheritance is that the extended type is compatible with the base type. Anything that works with objects or variables of the base type also work with objects of the extended type. A circle would have a radius in addition to an x and y position, a rectangle would have a width and height.
- dynamic binding – if we have a base shape class and derive circles and rectangles from it dynamic binding ensures that the correct method to calculate the area is called at run time.
- polymorphism- variables can therefore be polymorphic. Using the shape example we can therefore create an array of shapes, one may be a shape, one may be a circle and another may be a rectangle.

Extensible abstract data types with dynamically bound methods are often called classes. This is the terminology we will use in what follows.

26.3 Background Technical Material

We need to look more formally at a number of concepts so that we can actually do object oriented programming in Fortran. The following sections cover some of the introductory material we need, and are taken from the standard.

26.4 Type Declaration Statements

Every data object has a type and rank and may have type parameters and other attributes that determine the uses of the object. Collectively, these properties are the attributes of the object. The type of a named data object is normally specified explicitly in a type declaration statement. All of its attributes may be included in a type declaration statement or may be specified individually in separate specification statements.

26.4.1 TYPE

A TYPE type specifier is used to declare entities of a derived type. Section 1.3.147 of the standard defines it as follows:

- type: data type – named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values (4.1)

  A scalar entity of derived type is a structure.

26.4.2 CLASS

A polymorphic entity is a data entity that is able to be of differing types during program execution. The type of a data entity at a particular point during execution of a program is its dynamic type. The declared type of a data entity is the type that it is declared to have, either explicitly or implicitly.

A CLASS type specifier is used to declare polymorphic objects. The declared type of a polymorphic object is the specified type if the CLASS type specifier contains a type name.

26.4.3 Attributes

The additional attributes that may appear in the attribute specification of a type declaration statement further specify the nature of the entities being declared or specify restrictions on their use in the program.

26.4.3.1 Accessibility Attribute

The accessibility attribute specifies the accessibility of an entity via a particular identifier.

The following is taken from Sect. 5.3.2 of the Fortran 2008 standard.

- access-spec is PUBLIC or PRIVATE
- An access-spec shall appear only in the specification-part of a module.

Identifiers that are specified in a module or accessible in that module by use association have either the PUBLIC or PRIVATE attribute. Identifiers for which an access-spec is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement. Only identifiers that have the PUBLIC attribute in that module are available to be accessed from that module by use association.

26.4.4 Passed Object Dummy Arguments

Section 4.5.4.5 of the Fortran 2008 standard introduces the concept of passed object dummy argument. Here is an extract from the standard:

- A passed-object dummy argument is a distinguished dummy argument of a procedure pointer component or type-bound procedure. It affects procedure overriding (4.5.7.3) and argument association (12.5.2.2).
- If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object dummy argument.
- If neither PASS nor NOPASS is specified or PASS is specified without arg-name, the first dummy argument of a procedure pointer component or type-bound procedure is its passed-object dummy argument.
- If PASS (arg-name) is specified, the dummy argument named arg-name is the passed-object dummy argument of the procedure pointer component or named type-bound procedure.
- C456 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its length type parameters shall be assumed; it shall be polymorphic (4.3.1.3) if and only if the type being defined is extensible (4.5.7). It shall not have the VALUE attribute.
- NOTE 4.32: If a procedure is bound to several types as a type-bound procedure, different dummy arguments might be the passed-object dummy argument in different contexts.

The key here is that we are going to use the PASS and NOPASS attributes with type bound procedures – a component of object oriented programming in Fortran.

26.4.5 Derived Types and Structure Constructors

A derived type is a type that is not defined by the language but requires a type definition to declare its components. A scalar object of such a derived type is called a structure. Assignment of structures is defined intrinsically, but there are no intrinsic operations for structures. For each derived type, a structure constructor is available to provide values.

A derived-type definition implicitly defines a corresponding structure constructor that allows construction of values of that derived type.

26.4.6 Structure Constructors and Generic Names

A generic name may be the same as a type name. This can be used to emulate user-defined structure constructors for that type, even if the type has private components. The following example is taken from the standard to illustrate this.

```
module mytype_module
type mytype
private
complex value
logical exact
end type
interface mytype
module procedure int_to_mytype
end interface

! Operator definitions etc.

 …
contains
type(mytype) function int_to_mytype(i)
integer,intent(in) :: I
int_to_mytype%value = I
int_to_mytype%exact = .true.
end function
! Procedures to support operators etc.
…
end
```

26.4.7 Assignment

Execution of an assignment statement causes a variable to become defined or redefined. Simplistically

variable = expression

26.4.8 Intrinsic Assignment Statement

An intrinsic assignment statement is an assignment statement that is not a defined assignment statement. In an intrinsic assignment statement, variable shall not be polymorphic.

26.4.9 Defined Assignment Statement

A defined assignment statement is an assignment statement that is defined by a subroutine and a generic interface that specifies ASSIGNMENT (=).

26.4.10 Polymorphic Variables

Here is the definition of polymorphic taken from the standard.

- polymorphic – Able to be of differing types during program execution. An object declared with the CLASS keyword is polymorphic.

A polymorphic variable must be a pointer or allocatable variable. We will use allocatable variables to achieve polymorphism in our examples.

26.4.11 Executable Constructs Containing Blocks

The following are executable constructs that contain blocks:

- ASSOCIATE construct
- CASE construct
- DO construct
- IF construct
- SELECT TYPE construct

We will look at the ASSOCIATE construct and SELECT TYPE construct next.

26.4.12 ASSOCIATE Construct

The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. These named construct entities are associating entities. The names are associate names.

The following example illustrates an association with a derived-type variable.

```
ASSOCIATE ( XC => AX%B(I,J)%C )
XC%DV = XC%DV + PRODUCT(XC%EV(1:N))
end ASSOCIATE
```

26.4.13 Select Type Construct

The SELECT TYPE construct selects for execution at most one of its constituent blocks. The selection is based on the dynamic type of an expression. A name

is associated with the expression, in the same way as for the ASSOCIATE construct.

Quite a lot to take in! Let's illustrate the use of the above in some actual examples.

26.5 Example 1 – The Basic Shape Class

The code for the base shape class is given below.

- shape class data: integer variables x and y for the position.
- shape class methods: get and set for the x and y values, and moveto and draw.

We have used an include statement in the examples that follow to reduce code duplication.

We have used the default accessibility for the data and methods in the shape_ module.

```
module shape_module

  type shape_type
    integer :: x_ = 0
    integer :: y_ = 0

  contains
    procedure, pass(this) :: getx
    procedure, pass(this) :: gety
    procedure, pass(this) :: setx
    procedure, pass(this) :: sety
    procedure, pass(this) :: moveto
    procedure, pass(this) :: draw
  end type shape_type

contains

include "shape_module_common_code.f90"

end module shape_module
```

Here is the code in the include file.

```
  integer function getx(this)
    implicit none

    class (shape_type), intent (in) :: this
    getx = this%x_
  end function getx

  integer function gety(this)
    implicit none
```

```
    class (shape_type), intent (in) :: this
    gety = this%y_
  end function gety

  subroutine setx(this,x)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: x
    this%x_ = x
  end subroutine setx

  subroutine sety(this,y)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: y
    this%y_ = y
  end subroutine sety

  subroutine moveto(this,newx, newy)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: newx
    integer, intent (in) :: newy
    this%x_ = newx
    this%y_ = newy
  end subroutine moveto

  subroutine draw(this)
    implicit none

    class (shape_type), intent (in) :: this
    print *, ' x = ', this%x_
    print *, ' y = ', this%y_
  end subroutine draw
```

26.5.1 Key Points

Some of the key concepts are:

- We use a module as the organisational unit for the class.
- We use type and end type to contain the data and the procedures – called type bound procedures in Fortran terminology.
- The data in the base class is an x and y position.
- The type bound methods within the class are

  - getx and setx
  - gety and sety
  - draw
  - moveto
- We have used the default accessibility for the data and methods in the type.

Let us look at the code in stages.

```
module shape_module
```

The module is called shape_module

```
  type shape_type
```

The type is called shape_type

```
    integer :: x_ = 0
    integer :: y_ = 0
```

The data associated with the shape type are integer variables that are the x and y coordinates of the shape. We initialise to zero.

```
  contains
```

The type also contains procedures or methods.

```
  procedure, pass(this) :: getx
  procedure, pass(this) :: gety
  procedure, pass(this) :: setx
  procedure, pass(this) :: sety
  procedure, pass(this) :: moveto
  procedure, pass(this) :: draw
```

These are called type bound procedures in Fortran terminology. It is common in object oriented programming to have get and set methods for each of the data components of the type or object. We also have a moveto and draw method.

Each of these methods has the pass attribute. When a type bound procedure is called or invoked the object through which is invoked is normally passed as a hidden parameter. We have used the pass attribute to explicitly confirm the default behaviour of passing the invoking object as the first parameter. We have also followed the convention in object oriented programming of using the word `this` to refer to the current object.

```
  end type shape_type
```

This is the end of the type definition.

```
contains
```

The module then contains the actual implementation of the type bound procedures. We will look at a couple of these.

```
  integer function getx(this)
    implicit none
    class (shape_type), intent (in) :: this
```

```
    getx = this%x_
  end function getx
```

As we stated earlier it is common in object oriented programming to have get and set methods for each data item in an object. This function implements the getx method. The first argument is the current object, referred to as this. We then have the type declaration for this parameter. We declare the variable using class rather than type as we want the variable to be polymorphic. The rest of the function is self explanatory.

```
  subroutine setx(this,x)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: x
    this%x_ = x
  end subroutine setx
```

The setx procedure is a subroutine. It takes two parameters, the current object and the new x value. Again we use the class declaration mechanism as we want the variable to be polymorphic.

Here is a program to test the above class out.

```
program test_ch2601
  use shape_module
  implicit none

  type (shape_type) :: s1 = shape_type(10,20)

  integer :: x1 = 100
  integer :: y1 = 200

  print *, ' get '
  print *, s1%getx(), ' ', s1%gety()
  print *, ' draw '
  call s1%draw()
  print *, ' moveto '
  call s1%moveto (x1,y1)
  print *, ' draw '
  call s1%draw()
  print *, ' set '
  call s1%setx(99)
  call s1%sety(99)
  print *, ' draw'
  call s1%draw()
end program test_ch2601
```

The first statement of interest is the use statement, where we make available the shape_module to the test program. The next statement of interest is

```
  type (shape_type) :: s1 = shape_type(10,20)
```

We then have a type declaration for the variable s1. We also have the use of what Fortran calls a structure constructor `shape_type` to provide initial values to the x and y positions. The term constructor is used in other object oriented programming languages, e.g. C++, Java, C#. It has the same name as the type or class and is created automatically for us by the compiler in this example.

The

```
print *, s1%getx(), ' ', s1%gety()
```

statement prints out the x and y values for the object s1. We use the standard % notation that we used in derived types, to separate the components of the derived types. If one looks at the implementation of the getx function and examines the first line, repeated below

```
integer function getx(this)
```

how we refer to the current object, s1, through the syntax s1%getx(). The following call:

```
call s1%draw()
```

shows how to invoke the draw method for the s1 object, using the s1%draw() syntax. The first line of the draw subroutine

```
subroutine draw(this)
```

shows how the current object is passed as the first argument.

Here is the output from the Nag compiler.

```
d:\document\fortran\newbook\examples\ch26\ex01>nag_test_
ch2601.exe

 get
10    20
draw
x =   10
y =   20
moveto
draw
x =   100
y =   200
set
draw
x =   99
y =   99
```

26.5.2 Notes

In this example we have not used the public, private or protected attributes on the data or methods, we have just accepted the default Fortran accessibility behaviour. This means that we can use the compiler provided structure constructor `shape_type ()` as shown below

```
type (shape_type) :: s1 = shape_type(10,20)
```

in the type declaration to provide initial values, as they are public by default.

We have direct access to the data and methods as they are public by default in Fortran. This is often not a good idea for the data, as it is possible to makes changes to the data anywhere in the program. The next example makes the data private.

26.5.3 Example 1 with Private Data

Here is the modified base class. This example will now not compile as the default compiler provided structure constructor does not have access to the private data.

```
module shape_module

  type shape_type
    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains
    procedure, pass(this) :: getx
    procedure, pass(this) :: gety
    procedure, pass(this) :: setx
    procedure, pass(this) :: sety
    procedure, pass(this) :: moveto
    procedure, pass(this) :: draw
  end type shape_type

contains

include "shape_module_common_code.f90"

end module shape_module
```

Here is the same test program as in the first example.

```
program test_ch2602
  use shape_module
  implicit none
```

```
  type (shape_type) :: s1 = shape_type(10,20)

  integer :: x1 = 100
  integer :: y1 = 200

  print *, ' get '
  print *, s1%getx(), ' ', s1%gety()
  print *, ' draw '
  call s1%draw()
  print *, ' moveto '
  call s1%moveto(x1,y1)
  print *, ' draw '
  call s1%draw()
  print *, ' set '
  call s1%setx(99)
  call s1%sety(99)
  print *, ' draw'
  call s1%draw()
end program test_ch2602
```

Here is the output from trying to compile this example.

```
d:\document\fortran\newbook\examples\ch26\ex01>nagfor
-f2003 ch2602.f90 test_ch2
602.f90 -o nag_test_ch2602.exe
NAG Fortran Compiler: Release 5.2(722)
Evaluation trial version of NAG Fortran Compiler
Release 5.2(722)
ch2602.f90:
[NAG Fortran Compiler normal termination]
test_ch2602.f90:
Error: test_ch2602.f90, line 5: Constructor for type
SHAPE_TYPE which has PRIVATE component X_
Errors in declarations, no further processing for
TEST_CH2602
[NAG Fortran Compiler error termination, 1 error]

d:\document\fortran\newbook\examples\ch26\ex01>
```

An earlier solution to this type of problem can be found in the date class in Chap. 22, where we provide our own structure constructor `date_()`. Most object oriented programming languages provide the ability to use the same name as a class as a constructor name even if the data is private. Fortran 2003 provides another solution to this problem. In the example below we will provide our own structure constructor inside an interface.

26.5.4 Solution 1 with an Interface to Use the Class Name for the Structure Constructor

Here is the modified base class.

```
module shape_module

  type shape_type
    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains
    procedure, pass(this) :: getx
    procedure, pass(this) :: gety
    procedure, pass(this) :: setx
    procedure, pass(this) :: sety
    procedure, pass(this) :: moveto
    procedure, pass(this) :: draw
  end type shape_type

  interface shape_type
    module procedure shape_type_constructor
  end interface

contains
  type (shape_type) function
shape_type_constructor(x,y)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y

    shape_type_constructor%x_ = x
    shape_type_constructor%y_ = y
  end function shape_type_constructor

include "shape_module_common_code.f90"

end module shape_module
```

The key statements are

```
  interface shape_type
    module procedure shape_type_constructor
  end interface
```

which enables us to map a call or reference to shape_type (our structure constructor name) to our implementation of shape_type_constructor. Here is the implementation of this structure constructor.

```
  type (shape_type) function
shape_type_constructor(x,y)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y

    shape_type_constructor%x_ = x
    shape_type_constructor%y_ = y
  end function shape_type_constructor
```

The function is called shape_type_constructor hence we use this name to initialise the components of the type, and the function returns a value of type shape_type.

Here is the program to test the above out.

```
program ch2603
  use shape_module
  implicit none

  type (shape_type) :: s1
  integer :: x1 = 100
  integer :: y1 = 200

  s1 = shape_type(10,20)

  print *, ' get '
  print *, s1%getx(), ' ', s1%gety()
  print *, ' draw '
  call s1%draw()
  print *, ' moveto '
  call s1%moveto(x1,y1)
  print *, ' draw '
  call s1%draw()
  print *, ' set '
  call s1%setx(99)
  call s1%sety(99)
  print *, ' draw'
  call s1%draw()
end program ch2603
```

Note that in this example we cannot initialise `s1` at definition time using our own (user defined) structure constructor. This must now be done within the execution part of the program. This is a Fortran restriction, and makes it consistent with the rest of the language.

These examples illustrate some of the basics of object oriented programming in Fortran. To summarise

- the data in our class is private;
- access to the data is via get and set methods;

- the data and methods are within the derived type definition – the methods are called type bound procedures in Fortran terminology;
- we can use interfaces to provide user defined structure constructors, which have the same name as the class – this is a common practice in object oriented programming;
- we have used class to declare the variables within the type bound methods. We need to use class when we want to use polymorphic variables in Fortran.

26.5.5 Public and Private Accessibility

We have only made the internal data in the class private in the above example. There will be cases where some of the methods are only used within the class, in which case they can be made private.

26.6 Example 2 – Simple Inheritance

In this example we look at inheritance. We use the same base shape class and derive two classes from it – circle and rectangle.

A circle has a radius. This is the additional data component of the derived class. We also have get and set methods.

A rectangle has a width and height. These are the additional data components of the derived rectangle class. We also have get and set methods.

26.6.1 Base Shape Class

```
module shape_module

  type shape_type
    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains
    procedure, pass(this) :: getx
    procedure, pass(this) :: gety
    procedure, pass(this) :: setx
    procedure, pass(this) :: sety
    procedure, pass(this) :: moveto
    procedure, pass(this) :: draw
  end type shape_type

  interface shape_type
    module procedure shape_type_constructor
  end interface

contains
```

```
  type (shape_type) function &
shape_type_constructor(x,y)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y
    shape_type_constructor%x_ = x
    shape_type_constructor%y_ = y
  end function shape_type_constructor

include "shape_module_common_code.f90"

end module shape_module
```

The include file is the same as in the previous example.

26.6.2 Circle – Derived Type 1

Here is the first derived type.

```
module circle_module

use shape_module

type , extends(shape_type) :: circle_type

  integer , private :: radius_

  contains

  procedure , pass(this)  :: getradius
  procedure , pass(this)  :: setradius
  procedure , pass(this)  :: draw => draw_circle

end type circle_type

  interface circle_type
    module procedure circle_type_constructor
  end interface

  contains

  type (circle_type) function
circle_type_constructor(x,y,radius)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y
    integer, intent (in) :: radius
    call circle_type_constructor%setx(x)
```

```
    call circle_type_constructor%sety(y)
    circle_type_constructor%radius_=radius
  end function circle_type_constructor

  integer function getradius(this)
  implicit none
  class (circle_type) , intent(in) :: this
    getradius=this%radius_
  end function getradius

  subroutine setradius(this,radius)
  implicit none
  class (circle_type) , intent(inout) :: this
  integer , intent(in) :: radius
    this%radius_=radius
  end subroutine setradius

  subroutine draw_circle(this)
  implicit none
    class (circle_type), intent(in) :: this
    print *,' x = ' , this%getx()
    print *,' y = ' , this%gety()
    print *,' radius = ' , this%radius_
  end subroutine draw_circle

end module circle_module
```

Let us look more closely at the statements within this class. Firstly we have

```
module circle_module
```

which introduces our circle module.

We then

```
use shape_module
```

within this module to make available the shape class. The next statement

```
type , extends(shape_type) :: circle_type
```

is the key statement in inheritance. What this statement says is base our new circle_type on the base shape_type. It is an extension of the shape_type. We then have the additional data in our circle_type

```
    integer , private :: radius_
```

and the following additional type bound procedures.

```
    procedure , pass(this) :: getradius
    procedure , pass(this) :: setradius
    procedure , pass(this) :: draw => draw_circle
```

and we have the simple get and set methods for the radius, and a type specific draw method for our circle_type. It is this method that will be called when drawing with a circle, rather than the draw method in the base shape_type.

We then have an interface

```
interface circle_type
  module procedure circle_type_constructor
end interface
```

to provide us with our own user defined structure constructor for our circle_type.

As has been stated earlier it is common practice in object oriented programming to use the same name as the type for constructors.

We then have the implementation of the constructor.

```
  type (circle_type) function
circle_type_constructor(x,y,radius)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y
    integer, intent (in) :: radius

    call circle_type_constructor%setx(x)
    call circle_type_constructor%sety(y)
    circle_type_constructor%radius_=radius
  end function circle_type_constructor
```

Note that we use the setx and sety methods to provide initial values to the x and y values. They are private in the base class so we need to use these methods.

We can directly initialise the radius as this is a data component of this class, and we have access to it.

We next have the get and set methods for the radius.

Finally we have the implementation for the draw circle method.

```
  subroutine draw_circle(this)
  implicit none
    class (circle_type), intent(in) :: this
    print *,' x = ' , this%getx()
    print *,' y = ' , this%gety()
    print *,' radius = ' , this%radius_
  end subroutine draw_circle
```

Notice again that we use the getx and gety methods to access the x and y private data from the base shape class.

26.6.3 Rectangle – Derived Type 2

Here is the code for the second derived type.

```
module rectangle_module

use shape_module

type , extends(shape_type) :: rectangle_type

  integer , private :: width_
  integer , private :: height_

  contains

  procedure , pass(this) :: getwidth
  procedure , pass(this) :: setwidth
  procedure , pass(this) :: getheight
  procedure , pass(this) :: setheight
  procedure , pass(this) :: draw => draw_rectangle

end type rectangle_type

  interface rectangle_type
    module procedure rectangle_type_constructor
  end interface

contains
  type (rectangle_type) function
rectangle_type_constructor(x,y,width,height)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y
    integer, intent (in) :: width
    integer, intent (in) :: height

    call rectangle_type_constructor%setx(x)
    call rectangle_type_constructor%sety(y)
    rectangle_type_constructor%width_ = width
    rectangle_type_constructor%height_ = height
  end function rectangle_type_constructor

  integer function getwidth(this)
  implicit none
  class (rectangle_type) , intent(in) :: this
    getwidth=this%width_
  end function getwidth

  subroutine setwidth(this,width)
  implicit none
  class (rectangle_type) , intent(inout) :: this
```

```
  integer , intent(in) :: width
    this%width_=width
  end subroutine setwidth

  integer function getheight(this)
  implicit none
  class (rectangle_type) , intent(in) :: this
    getheight=this%height_
  end function getheight

  subroutine setheight(this,height)
  implicit none
  class (rectangle_type), intent(inout) :: this
  integer , intent(in) :: height
    this%height_=height
  end subroutine setheight

  subroutine draw_rectangle(this)
  implicit none
    class (rectangle_type), intent(in) :: this
    print *,' x = ' , this%getx()
    print *,' y = ' , this%gety()
    print *,' width = ' , this%width_
    print *,' height = ' , this%height_

  end subroutine draw_rectangle

end module rectangle_module
```

The code is obviously very similar to that of the first derived type.

26.6.4 Simple Inheritance Test Program

Here is a test program that illustrates the use of the shape type, circle type and rectangle type.

```
program ch2604

use shape_module
use circle_module
use rectangle_module

implicit none

type (shape_type)       :: vs
type (circle_type)     :: vc
type (rectangle_type) :: vr
```

```
  vs = shape_type(10,20)
  vc = circle_type(100,200,300)
  vr = rectangle_type(1000,2000,3000,4000)

  print *,' get '

  print *,' shape      ', vs%getx(),' ',vs%gety()
  print *,' circle     ', vc%getx(),' ',vc%gety(),'
radius = ',vc%getradius()
  print *,' rectangle ', vr%getx(),' ',vr%gety(),'
width  = ',vr%getwidth(),' height ',vr%getheight()

  print *,' draw '

  call vs%draw()
  call vc%draw()
  call vr%draw()

  print *,' set '

  call vs%setx(19)
  call vs%sety(19)

  call vc%setx(199)
  call vc%sety(199)
  call vc%setradius(199)

  call vr%setx(1999)
  call vr%sety(1999)
  call vr%setwidth(1999)
  call vr%setheight(1999)

  print *,' draw '

  call vs%draw()
  call vc%draw()
  call vr%draw()

end program ch2604
```

The first statements of note are

```
use shape_module
use circle_module
use rectangle_module
```

which make available the shape, circle and rectangle types within the program. The following statements

```
type (shape_type)        :: vs
type (circle_type)      :: vc
type (rectangle_type) :: vr
```

declare vs, vc and vr to be of type shape, circle and rectangle respectively. The following three statements

```
vs = shape_type(10,20)
vc = circle_type(100, 200, 300)
vr = rectangle_type(1000,2000, 3000, 4000)
```

call the three user defined structure constructor functions.

We then use the get functions to print out the values of the private data in each object.

```
print *,' shape      ', vs%getx(),' ',vs%gety()
print *,' circle     ', vc%getx() ,' ', &
  vc%gety(), 'radius = ',vc%getradius()
 print *,' rectangle ', vr%getx(),' ',vr%gety(),&
'width = ',vr%getwidth(),' height ',vr%getheight()
```

We then call the draw method for each type.

```
 call vs%draw()
 call vc%draw()
 call vr%draw()
```

and the appropriate draw method is called for each type.

We finally call the set functions for each variable and repeat the calls to the draw methods.

The draw methods in the derived types override the draw method in the base shape class.

26.7 Example 3 – Polymorphism and Dynamic Binding

An inheritance hierarchy can provide considerable flexibility in our ability to manipulate objects, whilst still taking advantage of static or compile time type checking. If we combine inheritance with polymorphism and dynamic binding we have a very powerful programming tool. We will illustrate this with a concrete example.

26.7.1 Base Shape Class

This is our base class. A polymorphic variable is a variable whose data type may vary at run time. It must be a pointer or allocatable variable, and it must be declared

using the class keyword. Our original base class declared variables using the class keyword from the beginning as we always intended to design a class that could be polymorphic.

We have had to make one change to the previous one. To make the polymorphism work we have had to provide our own assignment operator. So we have

```
  interface assignment (=)
    module procedure generic_shape_assign
  end interface
```

which means that our implementation of the procedure generic_shape_assign will replace the intrinsic assignment. Here is the actual implementation.

```
  subroutine generic_shape_assign(lhs,rhs)
    implicit none

    class (shape_type), intent (out), allocatable ::
lhs
    class (shape_type), intent (in) :: rhs

    allocate (lhs,source=rhs)
  end subroutine generic_shape_assign
```

In an assignment we obviously have

```
   left_hand_side = right_hand_side
```

and in our code we have variables lhs and rhs to clarify what is happening. We also have an enhanced form of allocation statement:

```
   allocate (lhs,source=rhs)
```

and the key is that the left hand side variable is allocated with the values and type of the right hand side variable. Here is the complete code.

```
module shape_module

  type shape_type
    integer, private :: x_ = 0
    integer, private :: y_ = 0

  contains
    procedure, pass(this) :: getx
    procedure, pass(this) :: gety
    procedure, pass(this) :: setx
    procedure, pass(this) :: sety
    procedure, pass(this) :: moveto
    procedure, pass(this) :: draw
  end type shape_type

  interface shape_type
```

```
    module procedure shape_type_constructor
  end interface

  interface assignment (=)
    module procedure generic_shape_assign
  end interface

contains

  type (shape_type) function &
shape_type_constructor(x,y)
    implicit none
    integer, intent (in) :: x
    integer, intent (in) :: y

    shape_type_constructor%x_ = x
    shape_type_constructor%y_ = y
  end function shape_type_constructor

  integer function getx(this)
    implicit none

    class (shape_type), intent (in) :: this
    getx = this%x_
  end function getx

  integer function gety(this)
    implicit none

    class (shape_type), intent (in) :: this
    gety = this%y_
  end function gety

  subroutine setx(this,x)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: x
    this%x_ = x
  end subroutine setx

  subroutine sety(this,y)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: y
    this%y_ = y
  end subroutine sety
```

```
  subroutine moveto(this,newx,newy)
    implicit none

    class (shape_type), intent (inout) :: this
    integer, intent (in) :: newx
    integer, intent (in) :: newy
    this%x_ = newx
    this%y_ = newy
  end subroutine moveto

  subroutine draw(this)
    implicit none

    class (shape_type), intent (in):: this
    print *, ' x = ', this%x_
    print *, ' y = ', this%y_
  end subroutine draw

  subroutine generic_shape_assign(lhs,rhs)
    implicit none

    class (shape_type), intent (out), allocatable ::
lhs
    class (shape_type), intent (in) :: rhs

    allocate (lhs,source=rhs)
  end subroutine generic_shape_assign

end module shape_module
```

26.7.2 Circle – Derived Type 1

The circle code is the same as before.

26.7.3 Rectangle – Derived Type 2

The rectangle code is as before.

26.7.4 Shape Wrapper Module

As was stated earlier a polymorphic variable must be a pointer or allocatable variable. We have chosen to go the allocatable route. The following is a wrapper routine to allow us to have a derived type whose types can be polymorphic.

```
module shape_wrapper_module

  use shape_module
  use circle_module
  use rectangle_module

  type shape_wrapper
    class (shape_type), allocatable :: x
  end type shape_wrapper

end module shape_wrapper_module
```

So in this case x can be of shape_type or of any type derived from shape_type. Don't panic if this isn't clear at the moment, the complete program should help out!

26.7.5 Display Subroutine

This is the key subroutine in this example. We can pass into this routine an array of type shape_wrapper. In the code so far we have variables of type

- shape_type
- circle_type
- rectangle_type

and we are passing in an array of elements and each element can be of any of these types, i.e. the shape_array is polymorphic.

We next statement of interest is

```
      call shape_array(i)%x%draw()
```

and at run time the correct draw method will be called. This is called dynamic binding. Here is the complete code.

```
module display_module

contains

  subroutine display(n_shapes,shape_array)
    use shape_wrapper_module
    implicit none
    integer, intent (in) :: n_shapes
    type (shape_wrapper), dimension (n_shapes) ::
shape_array
    integer :: i

    do i = 1, n_shapes
      call shape_array(i) %x%draw()
    end do
  end subroutine display

end module display_module
```

26.7.6 Test Program

We now have the complete program that illustrates polymorphism and dynamic binding in action.

```
program ch2605
   use shape_module
   use circle_module
   use rectangle_module
   use shape_wrapper_module
   use display_module
   implicit none
   integer, parameter :: n = 6
   integer :: i

   type (shape_wrapper), dimension (n) :: s

   s(1) %x = shape_type(10,20)
   s(2) %x = circle_type(100,200,300)
   s(3) %x = rectangle_type(1000,2000,3000,4000)
   s(4) %x = s(1)%x
   s(5) %x = s(2)%x
   s(6) %x = s(3)%x

   print *,' calling display subroutine'

   call display(n,s)

   print *,' select type with get methods'

   do i=1,n
     select type ( t=>s(i) %x )
        class is (shape_type)
          print *,' x = ', t%getx(),&
            ' y = ',t%gety()
        class is (circle_type)
          print *,' x = ', t%getx(),&
            ' y = ',t%gety()
          print *,' radius = ', t%getradius()
        class is (rectangle_type)
          print *,' x = ', t%getx(),&
            ' y = ',t%gety()
          print *,' height = ', t%getheight()
          print *,' width = ', t%getwidth()
        class default
          print *,' do nothing'
     end select
   end do
```

```
  print *,' select type with set methods'

  do i=1,n
    select type ( t=>s(i) %x )
      class is (shape_type)
        call t%setx(19)
        call t%sety(19)
      class is (circle_type)
        call t%setx(199)
        call t%sety(199)
        call t%setradius(199)
      class is (rectangle_type)
        call t%setx(1999)
        call t%sety(1999)
        call t%setheight(1999)
        call t%setwidth(1999)
      class default
        print *,' do nothing'
    end select
  end do

  print *,' calling display subroutine'

  call display(n,s)

end program ch2605
```

Let us look at the key statements in more detail.

```
  type (shape_wrapper), dimension (n) :: s
```

This is the key declaration statement. S will be our polymorphic array. The following six assignment statements

```
  s(1) %x = shape_type(10,20)
  s(2) %x = circle_type(100,200,300)
  s(3) %x = rectangle_type(1000,2000,3000,4000)
  s(4) %x = s(1)%x
  s(5) %x = s(2)%x
  s(6) %x = s(3)%x
```

will call our own assignment subroutine to do the assignment. The allocation is hidden in the implementation. We then have

```
  call display(n,s)
```

which calls the display subroutine. The compiler at run time works out which draw method to call depending of the type of the elements in the shape_wrapper array.

Imagine now adding another shape type, let us say a triangle. We need to do the following

- inherit from the base shape type
- add the additional data to define a triangle
- add the appropriate get and set methods
- add a draw triangle method
- add a use statement to the shape_wrapper_module
- add a use statement to the main program

and we now can work with the new triangle shape type. The display subroutine is unchanged! We can repeat the above steps for any additional shape type we want.

Polymorphism and dynamic binding thus shorten our development and maintenance time, as it reduces the amount of code we need to write and test.

We then have an example of the use of the SELECT TYPE statement. The compiler determines the type of the elements in the array and then executes the matching block.

```
  do i=1,n
    select type ( t=>s(i) %x )
      class is (shape_type)
        print *,' x =         ',       t%getx(),' y =
',t%gety()
      class is (circle_type)
        print *,' x =         ',       t%getx(),' y =
 ',t%gety()
        print *,' radius =    ',       t%getradius()
      class is (rectangle_type)
        print *,' x =         ',       t%getx(),' y =
',t%gety()
        print *,' height =    ',       t%getheight()
        print *,' width =     ',       t%getwidth()
      class default
        print *,' do nothing'
    end select
end do
```

Now imagine adding support for the new triangle type. Anywhere we have select type constructs we have to add support for our new triangle shape. There is obviously more work involved when we use the select type construct in our polymorphic code. However some problems will be amenable to polymorphism and dynamic binding, others will require the explicit use of select type statements. This example illustrates the use of both.

26.7.7 Program Output

Try running the program. Here is the output from one compiler.

```
d:\document\fortran\newbook\examples\ch26\ex03>ch2636
  calling display subroutine
  x =  10
  y =  20
  x =  100
  y =  200
  radius = 300
  x =  1000
  y =  2000
  width =   3000
  height =   4000
  x =  10
  y =  20
  x =  100
  y =  200
  radius =   300
  x =  1000
  y =  2000
  width =   3000
  height =   4000
  select type with get methods
  x =           10  y =  20
  x =           100  y =  200
  radius =     300
  x =           1000  y =  2000
  height =     4000
  width =      3000
  x =           10  y =  20
  x =           100  y =  200
  radius =     300
  x =           1000  y =  2000
  height =     4000
  width =      3000
  select type with set methods
  calling display subroutine
  x =  19
  y =  19
```

```
x =  199
y =  199
radius =   199
x =  1999
y =  1999
width =   1999
height =   1999
x =  19
y =  19
x =  199
y =  199
radius = 199
x =  1999
y =  1999
width =   1999
height =   1999
```

26.8 Summary

This chapter has introduced some of the essentials of object oriented programming. The first example looked at object oriented programming as an extension of basic data structuring. We used type bound procedures to implement our shape class. We used methods to access the internal data of the shape object.

The second example looked at simple inheritance. We saw in this example how we could reuse the methods from the base class and also add new data and methods specific to the new shapes – circles and rectangles.

The third example then looked at how to achieve polymorphism in Fortran. We could then create arrays of our base type and dynamically bind the appropriate methods at run rime. Dynamic binding is needed when multiple classes contain different implementations of the same method, i.e. to ensure in the following code

```
call shape_array(i) %x%draw()
```

that the correct draw method is invoked on the shape object.

26.9 Problems

1. Compile and run all of the examples in this chapter with your compiler. At the time of writing this book not all compilers compiled and ran these examples. This situation will improve with time. If your compiler doesn't complain!
2. Add a triangle type to the simple inheritance example.
3. Add a triangle type to the polymorphic example.

26.10 Bibliography

Birtwistle, G.M., Dahl, O.J., Myhrhaug, B., Nygaard, K.: SIMULA BEGIN. Chartwell-Bratt Ltd, Bromley (1979)

Goldberg, A., Robson, D.: Smalltalk-80. The Language and Its Implementation. Addison Wesley, Reading (1983)

ISO/IEC 1539–1:2010 Information technology – Programming languages – Fortran – Part 1: Base language

Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Upper Saddle River (1997)

Mossenbeck, H.: Object-Orientated Programming in Oberon-2. Springer-Verlag, Berlin/New York (1995)

Stroustroup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley, Reading (1997)

Wiener, R.: Software Development Using Eiffel. Prentice Hall, Englewood Cliffs (1995)

Chapter 27
Introduction to Parallel Programming

'Can you do addition?' the White Queen asked. 'What's one and one and one and one and one and one and one and one and one?'
'I don't know' said Alice. 'I lost count.'
'She can't do addition,' the Red Queen interrupted.

Lewis Carroll, Through the Looking Glass and What Alice Found There.

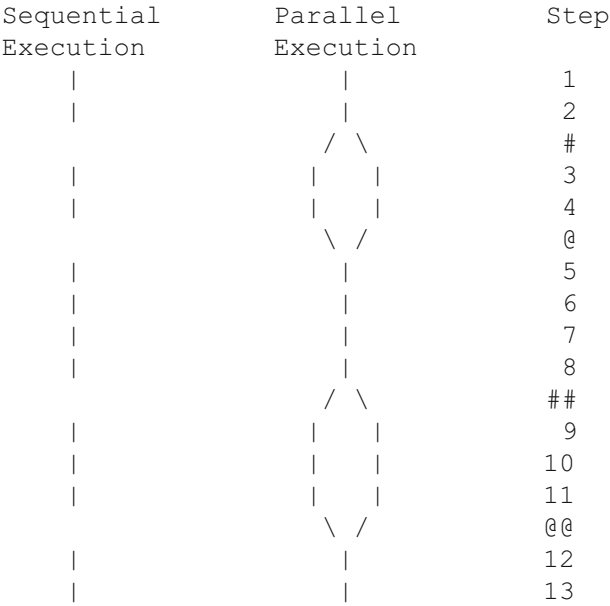
Aim

The aims of this chapter is to provide a short introduction to parallel programming.

27.1 Introduction

Parallel programming involves breaking a program down into parts that can be executed concurrently. Here is a simple diagram to illustrate the idea.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_27, © Springer-Verlag London Limited 2012



On the left hand side we have a sequential program and this steps through linearly from beginning to end. The right hand side has the same program that has been partially parallelised. There are two parallel regions and the work here is now shared between two processes or threads. At each parallel part of the program we have the following

	Parallel Region 1	Parallel Region 2
Set up cost	Step #	Step ##
Parallel section	Steps 3,4	Steps 9,10,11
Synchronisation cost	Step @	Step @@

The theory is that the overall run time of the program will have been reduced or we will have been able to solve a larger problem by parallelising our code. In the above example we have divided the work between two processes or threads. Here are some details of a range of processors which support multiple cores.

Processor		Cores
AMD Phenom II	X6	6
Intel Core i7	920	4 *2
Intel Core i7	2600K	4 *2
AMD Opteron	Shanghai	4
	Istanbul	6
	Magny Cours	8
	Magny Cours	12

Visit the AMD and Intel sites for up to date information.

There are several ways of doing parallel programming, and this chapter will look at three ways of doing this in Fortran. There are a common set of concepts and terminology that are useful to know about, whichever method we use, and we will cover these first.

27.2 Parallel Computing Classification

Parallel computing is often classified by the way the hardware supports parallelism. Two of the most common are:

- multi-processor and multi-core computers having multiple processing elements within a single system
- clusters or grids with multiple computers connected to work together.

Modern large systems are increasingly hybrids of the two above.

27.3 Amdahl's Law

Amdahl's law is a simple equation for the speedup of a program when parallelised. It assumes that the problem size remains the same when parallelized.

In the equation below

- P is the proportion of the program that can be parallelised
- (1-P) is the serial proportion
- N is the number of processors

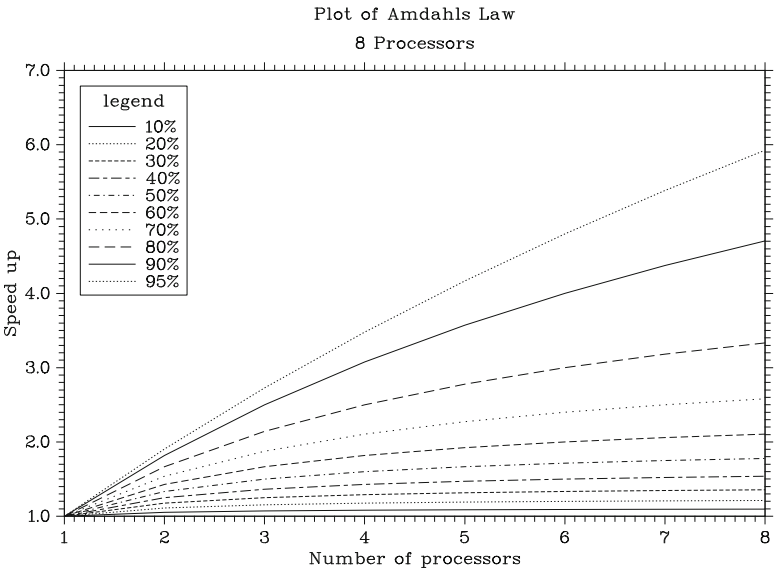
and we have

- speedup = 1/((1-P) + P/N)

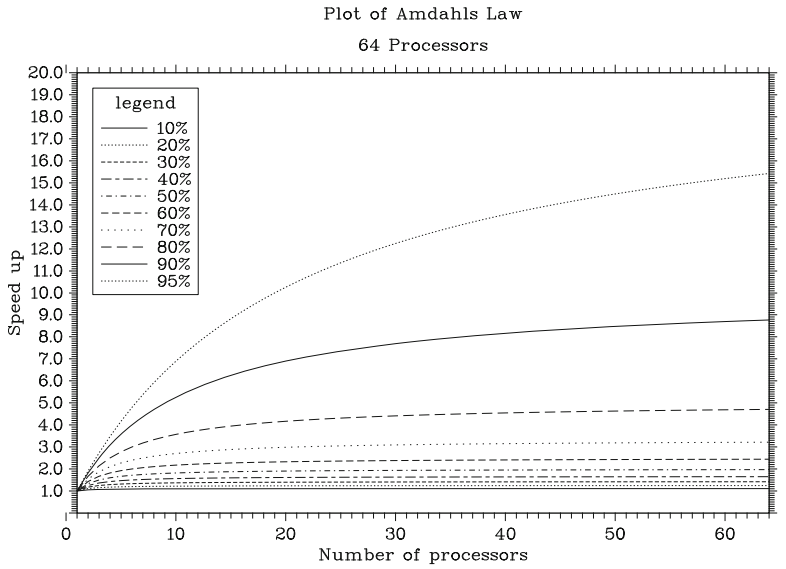
We have included a couple of graphs to illustrate the above. We have used the dislin graphics library to do the plots. It is available from

http://www.mps.mpg.de/dislin/

27.3.1 Amdahl's Law Graph 1–8 Processors or Cores



27.3.2 Amdahl's Law Graph 2–64 Processors or Cores



Here is the source code for one of the programs. It is for the 8 processor version.

```
program ch2701
use dislin
implicit none
integer :: i,j
! Total number of processors and hence data points
integer , parameter     :: nprocs = 8
! Number of percentage values from
! 10% -> 90%  9
! 95%          1
! Total      10
integer , parameter     :: nn = 10
real , dimension(nn)    :: pp=(/
0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.95 /)
real , dimension(nprocs)     :: x
real , dimension(nprocs)     :: y
real , dimension(nprocs,nn) :: ydata
integer :: nx
integer :: ny
character*30 cbuf

  do i=1,nprocs
    x(i)=real(i)
  end do

! Amdahl calculations. Store in 2 d array and then
! assign to 1 d array for plotting.

  do i=1,nprocs
     do j=1,nn
        ydata(i,j) = 1 / ( ( 1-pp(j) ) + pp(j) / I )
     end do
  end do

! Write the data to a file for verification purposes

  open (unit=10,file='amdahl_table_08.txt')
  do i=1,nprocs
    write(unit=10,fmt=100) x(i),ydata(i,1:nn)
    100 format(11(f7.2,2x))
  end do
  close(10)

  call disini
  call complx
```

```
  call axspos(450,1800)
  call axslen(2200,1400)
  call name('Number of processors','x')
  call name('Speed up','y')
  call titlin('Plot of Amdahls Law',1)
  call titlin('8 Processors',3)
  call labdig(-1,'x')
  call ticks(10,'xy')
  call graf(1.0,8.0,1.0,1.0,1.0,7.0,1.0,1.0)
  call title
  call xaxgit
  call chncrv('line')

! Plot the curves. Copy from 2 d array to 1 d array
! before the call to curve.

  do i=1,nn
    y=ydata(1:nprocs,i)
    call curve(x,y,nprocs)
  end do

  call legini(cbuf,10,3)

! Coordinates of the start of the legend
! for the curves.

  nx=500
  ny=450
  call legpos(nx,ny)
  call leglin(cbuf,'10%',1)
  call leglin(cbuf,'20%',2)
  call leglin(cbuf,'30%',3)
  call leglin(cbuf,'40%',4)
  call leglin(cbuf,'50%',5)
  call leglin(cbuf,'60%',6)
  call leglin(cbuf,'70%',7)
  call leglin(cbuf,'80%',8)
  call leglin(cbuf,'90%',9)
  call leglin(cbuf,'95%',10)
  call legtit('legend')
  call legend(cbuf,3)
  call disfin
end program ch2701
```

We use the dislin graphics library in this example. More information about the dislin software can be found in Chap. 33.

27.4 Gustafson's Law

Gustafson's Law is often seen as a contradiction of Amdahl's Law. Simplistically it states that programmers solve larger problems when parallelising programs.

The equation for Gustafson's Law is given below.

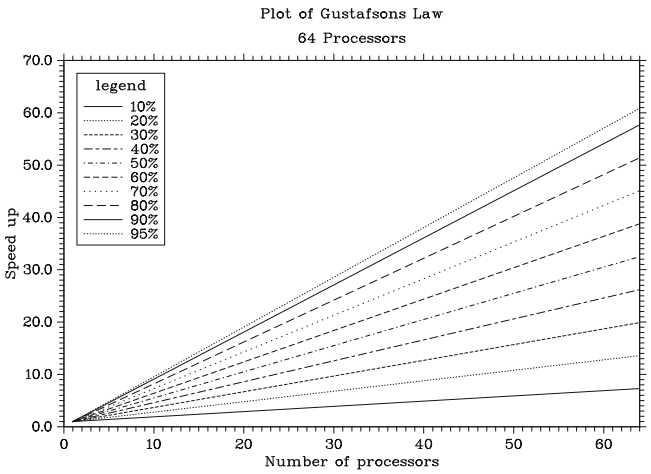
- N is the number of processors
- Serial is the proportion that remains serial

and

- Speedup(N) = N – serial * (N – 1)

We have again included a graph to illustrate the above.

27.4.1 Gustafson's Law Graph 1–64 Processors or Cores



Here is the source code for the program. This is for the 64 processor version.

```
program ch2702
use dislin
implicit none
integer :: i,j
! Total number of processors and hence data points
integer , parameter     :: nprocs = 64
! Number of percentage values from
! 10% -> 90%  9
! 95%          1
```

```
! Total      10
integer , parameter      :: nn = 10
real , dimension(nn)     :: pp=&
(/0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.95 /)
real , dimension(nprocs)     :: x
real , dimension(nprocs)     :: y
real , dimension(nprocs,nn) :: ydata
integer :: nx
integer :: ny
character*30 cbuf

  do i=1,nprocs
    x(i)=real(i)
  end do

! gustafson calculations. Store in 2 d array and then
! assign to 1 d array for plotting.

  do i=1,nprocs
    do j=1,nn
        ydata(i,j) = I - (1-pp(j)) * (I-1)
    end do
  end do

! Write the data to a file for verification purposes

  open (unit=10,file='gustafson_table.txt')
  do i=1,nprocs
    write(unit=10,fmt=100) x(i),ydata(i,1:nn)
    100 format(11(f7.2,2x))
  end do
  close(10)

  call disini
  call complx
  call axspos(450,1800)
  call axslen(2200,1400)
  call name('Number of processors','x')
  call name('Speed up','y')
  call titlin('Plot of Gustafsons Law',1)
  call titlin('64 Processors',3)
  call labdig(-1,'x')
  call ticks(10,'xy')
  call graf(0.0,64.0,0.0,10.0,0.0,70.0,0.0,10.0)
  call title
```

```
  call xaxgit
  call chncrv('line')

! Plot the curves. Copy from 2 d array to 1 d array
! before the call to curve.

  do i=1,nn
    y=ydata(1:nprocs,i)
    call curve(x,y,nprocs)
  end do

  call legini(cbuf,10,3)

! Coordinates of the start of the legend
! for the curves.

  nx=500
  ny=450
  call legpos(nx,ny)
  call leglin(cbuf,'10%',1)
  call leglin(cbuf,'20%',2)
  call leglin(cbuf,'30%',3)
  call leglin(cbuf,'40%',4)
  call leglin(cbuf,'50%',5)
  call leglin(cbuf,'60%',6)
  call leglin(cbuf,'70%',7)
  call leglin(cbuf,'80%',8)
  call leglin(cbuf,'90%',9)
  call leglin(cbuf,'95%',10)
  call legtit('legend')
  call legend(cbuf,3)
  call disfin
end program ch2702
```

The programs are obviously very similar!

27.5 Memory Access

Memory access times fall into two main categories that are of interest in parallel computing

- uma – uniform memory access. Each element of main memory can be accessed with the same latency and bandwidth. Multi-processor and multi-core computers typically have this behaviour.

- numa – non uniform memory access. Distributed memory systems have non-uniform memory access. Clusters or grids with multiple computers connected to work together have this behaviour.

27.6 Cache

Modern processors have a memory hierarchy. They typically have two or more levels:

- main memory
- cpu memory

and there is a speed and cost link. Main memory is cheap and relatively slow in comparison to the cpu memory.

The cpu memory or cache is used to reduce the effective access time to memory. If the information that the program requires is in the cpu cache then the average latency of memory accesses will be closer to the cache latency than to the latency of main memory. Getting high performance from a computer normally means writing cache friendly programs. This means that the data and instructions that the program needs are already in the cache and don't need to be accessed from the much slower main memory.

In a multi-core and multi-cpu system each core and cpu will have their own memory or cache. This introduces the problem of cache coherency – i.e. the consistency of data stored in local caches compared to the data in the common shared memory. This problem must obviously be addressed when doing parallel programming.

27.7 Bandwidth and Latency

Bandwidth is the rate at which data can be transferred. Latency is the start up time for a data transfer. We normally want a high bandwidth and low latency. Here are some figures for several interconnects.

	MPI bandwidth or maximum theoretical GB/s	Latency µs
Gigabit ethernet	0.125	~100
Infiniband	1.3	4.0
Myrinet 10-G	1.2	2.1
Quadrics QsNet II	0.9	2.7
Cray SeStar2	2.1	4.5

27.8 Flynn's Taxonomy

Flynn's taxonomy is an old, but still widely used, classification scheme for computer architecture.

- Single Instruction, Single Data stream (SISD) A sequential computer which exploits no parallelism in either the instruction or data streams. Term rarely used.
- Single Instruction, Multiple Data streams (SIMD) A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or GPU.
- Multiple Instruction, Single Data stream (MISD) Multiple instructions operate on a single data stream. Term rarely used.
- Multiple Instruction, Multiple Data streams (MIMD) Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space. Essentially separate computers working together to solve a problem.

We also have the term

- Single Program Multiple Data – An identical program executes on a MIMD computer system. Conditional statements in the code mean that different parts of the program execute on each system.

27.9 Consistency Models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

27.10 Threads and Threading

In computing a thread of execution is often regarded as the smallest unit of processing that can be scheduled by an operating system. The implementation of threads and processes generally varies with operating system.

27.11 Threads and Processes

From a strict computer science point of view threads and processes are different. However when looking simply at parallel programming the term can often be used interchangeably. In the following we use the term thread.

27.12 Data Dependencies

A data dependency is when one statement in a program depends on a calculation from a previous statement. This will obviously hinder parallelism.

27.13 Race Conditions

Race conditions can occur in programs when separate threads depend on a shared state or variable.

27.14 Mutual Exclusion – Mutex

A mutex is a programming construct that is used to allow multiple threads to share a resource. The sharing is not simultaneous. One thread will acquire the mutex and then lock the other threads from accessing it until it has completed.

27.15 Monitors

In concurrent programming, a monitor is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared with code that may be executed in parallel.

27.16 Locks

In computing a lock is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies.

27.17 Synchronization

The concept of synchronisation is often split into process and data synchronisation.

In process synchronisation several processes or threads come together at a certain part of a program.

Data synchronisation is concerned with keeping data consistent.

27.18 Granularity and Types of Parallelism

Granularity is a useful concept in parallel programming. A common classification is

- Fine-grained – a lot of small components, larger amounts of communication and synchronisation
- Coarse-grained – a small number of larger components, hence smaller amounts of communication and less synchronisation

The terms are of course relative.

We also have the concept of

- Embarrassingly parallel – very little effort is required to partition the task and there is little or no communication and synchronisation.

A simple example of this would be a graphics processor processing individual pixels.

27.19 Partitioned Global Address Space – PGAS

PGAS is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each processor. The PGAS model is the basis of Unified Parallel C, Coarray Fortran, Titanium, Fortress, Chapel and X10.

27.20 Fortran and Parallel Programming

Most Fortran compilers now offer support for parallel programming. We next provide a brief coverage of three methods

- MPI – Message Passing Interface
- OpenMP – Open Multi-Processing
- CoArray Fortran

Subsequent chapters look at simple examples using each method.

27.21 MPI

MPI started with a meeting that was held at the Supercomputing 92 conference. The attendants agreed to develop and implement a common standard for message passing. The first MPI standard, called MPI-1 was completed in May 1994. The second MPI standard, MPI-2, was completed in 1998.

MPI is effectively a library of C and Fortran callable routines. It has become widely used and is available on a number of platforms. Some useful web addresses are given below.

The first is hosted at Argonne National Laboratory.

http://www.mcs.anl.gov/research/projects/mpi/

MPI was designed by a broad group of parallel computer users, vendors, and software writers. These included

- Vendors – IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
- Library writers – PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
- Companies – ARCO, Convex, Cray Research, IBM, Intel, KAI, Meiko, NAG, nCUBE, Parasoft, Shell, TMC
- Laboratories – ANL, GMD, LANL, LLNL, NOAA, NSF, ORNL, PNL, Sandia, SDSC, SRC
- Universities – UC Santa Barbara, Syracuse University, Michigan State University, Oregon Grad Inst, University of New Mexico, Mississippi State University, U of Southampton, University of Colorado, Yale University, University of Tennessee, University of Maryland, Western Michigan University, University of Edinburgh, Cornell University, Rice University, University of San Francisco

So whilst MPI is not a formal standard like Fortran, C or C++, its development has involved quite a wide range of people.

The following site has details of MPI meetings.

http://meetings.mpi-forum.org/

The steering committee (as of April 2011) and affiliations are given below

- Jack Dongarra – Computer Science Department, University of Tennessee
- Al Geist – Group Leader, Computer Science Research Group, Oak Ridge National Laboratory
- Richard Graham
- Bill Gropp – Computer Science Department, University of Illinois Urbana-Champaign
- Andrew Lumsdaine – Computer Science Department, Indianna University
- Ewing Lusk – Mathematics and Computer Science Division, Argonne National Laboratory
- Rolf Rabenseifner – High Performance Computing Center, Germany

The meeting coordinators and associated work areas as of April 2011 are given below

- Richard Graham, Convener and Meeting Chair
- Jeff Squyres, Meeting Secretary
- Bill Gropp, Ballots
- Rolf Rabenseifner, MPI 2.1
- Bill Gropp, MPI 2.2
- Torsten Hoefler, Andrew Lumsdaine, MPI 3.0 – Collective Communications

- Richard Graham, MPI 3.0 – Fault Tolerance
- Craig Rasmussen, MPI 3.0 – Fortran Bindings
- Bill Gropp and Rajeev Thakur, MPI 3.0 – Remote Memory Access
- Martin Schulz and Bronis de Supinski, MPI 3.0 – Tools support
- Pavan Balaji, MPI 3.0 – Hybrid Programming
- Anthony Skjellum, MPI 3.0 – Persistence
- Ron Brightwell – MPI 3.0 – Point-To-Point Communicationsand this provides an idea of the work currently in progress.

Another useful site is the Open MPI site.

http://www.open-mpi.org/

The following is taken from their site.

The Open MPI Project is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

Features implemented or in short-term development for Open MPI include:

- Full MPI-2 standards conformance
- Thread safety and concurrency
- Dynamic process spawning
- Network and process fault tolerance
- Support network heterogeneity
- Single library supports all networks
- Run-time instrumentation
- Many job schedulers supported
- Many OS's supported (32 and 64 bit)
- Production quality software
- High performance on all platforms
- Portable and maintainable
- Tunable by installers and end-users
- Component-based design, documented APIs
- Active, responsive mailing list
- Open source license based on the BSD license

Both sites provide free downloadable implementations.
Commercial implementations are available from

- Cray
- IBM
- Intel
- Microsoft

amongst others.

MPI is, at the time of writing, the dominant parallel programming method used in Fortran. MPI and Fortran currently account for over 80% of the code running on the HECToR Service in Edinburgh. HECToR is the UK's high-end computing resource, funded by the UK Research Councils. Visit

http://www.hector.ac.uk

for more information.

27.22 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface that supports shared memory multiprocessing programming in three main languages (C, C++, and Fortran) on a range of hardware platforms and operating systems. It consists of a set of compiler directives, library routines, and environment variables that determine the run time behaviour of a program.

The OpenMP Architecture Review Board (ARB) has published several versions

- October 1997 – OpenMP for Fortran 1.0. October the following year they released the C/C++ standard.
- 2000 – Fortran version
- 2005 – Fortran 2.5
- 2008 – OpenMP 3.0. Included in the new features in 3.0 is the concept of tasks and the task construct.
- 2011-OpenMP 3.1

A number of compilers from various vendors or open source communities implement the OpenMP API, including

- Absoft
- Cray
- gnu
- Hewlett Packard
- IBM
- Intel
- Lahey/Fujitsu
- Oracle/Sun
- PGI

The main OpenMP web site is:

http://www.openmp.org/

27.23 Coarray Fortran

Coarrays became part of Fortran in the 2008 standard. The original ideas came from work by Robert Numrich and John Reid in the 1990s. They are based on a single program multiple data model. A coarray Fortran program is interpreted as if it were duplicated several times and all copies execute asynchronously. Each copy has its own set of data objects and is termed an image. The array syntax of Fortran is extended with additional trailing subscripts in square brackets to provide a concise representation of references to data that is spread across images.

The syntax is architecture independent and may be implemented on:

- Distributed memory machines.
- Shared memory machines.
- Clustered machines.

Work is underway for additional Coarray functionality for the next standard.

27.24 Other Parallel Options

There are a number of additional parallel methods. They are covered for completeness.

27.24.1 PVM

Parallel Virtual Machine consists of a library and a run-time environment which allow the distribution of a program over a network of (even heterogeneous) computers. Visit

- http://www.epm.ornl.gov/pvm/
- http://www.netlib.org/pvm3/

for more details.

27.24.2 HPF

To quote their home page

- http://hpff.rice.edu/index.htm

'The High Performance Fortran Forum (HPFF), a coalition of industry, academic and laboratory representatives, works to define a set of extensions to Fortran 90 known collectively as High Performance Fortran (HPF). HPF extensions provide access to high-performance architecture features while maintaining portability across platforms.'

They also provide details of:

- Surveys of HPF compilers and tools.
- Currently available commercial HPF compilers.
- public domain HPF compilation systems.
- Research prototypes of HPF and HPF-related compilation systems.
- Mailing list.

27.25 Top 500 Supercomputers

Have a look at

- http://www.top500.org/

for a lot of links to supercomputing centres and information on parallel computing in general.

To see what can be done with all this processing power visit:

- http://www.met-office.gov.uk/

27.26 Summary

Fortran has long been one of the main languages used in parallel programming. This chapter has provided a brief coverage of some of the background to parallel programming in general, and Fortran in particular.

In the next three chapters we will look at a small number of programs that introduce some of the basic syntax of parallel programming with MPI, OpenMP and Coarray Fortran. We will also look at solving one problem serially and then solve it using the parallel features provided by MPI, OpenMP and Coarray Fortran. We provide timing details so that we can see the benefits that parallel solutions offer.

27.27 Bibliography

The ideas involved in parallel computing are not new and we've included a couple of references about computer hardware and operating systems, which provide information for the more inquisitive reader. Wikipedia is an on-line source of information in this area.

27.27.1 Computer Hardware

Baer, J.L.: Computer Systems Architecture. Computer Science Press, Rockville (1980)

The chapters on the memory hierarchy and memory management are old, but well written. Up to date hardware information can be found at most hardware vendor sites. A few are given below.

27.27.1.1 AMD

http://developer.amd.com/pages/default.aspx

27.27.1.2 IBM

http://www.ibm.com/products/us/en/

27.27.2 Intel

http://www.intel.com/en_UK/products/processor/index.htm

27.27.3 Computer Operating Systems

Deitel, H.M.: Operating Systems. Addison Wesley, Reading (1990)

Part two of the book (process management) has chapters on process concepts, asynchronous concurrent processes, concurrent programming and deadlock and indefinite postponement. The bibliographies at the end of each chapter are quite extensive.

27.27.4 Parallel Programming

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann, San Francisco
Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP. MIT Press, Cambridge
Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, Cambridge
Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann, San Francisco

Chapter 28
MPI – Message Passing Interface

In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Ada Lovelace

Aim

The aims of this chapter is to provide a short introduction to MPI programming in Fortran.

28.1 Introduction

Documents for the MPI standard are available from the MPI Forum. Their web address is

`http://www.mpi-forum.org`

If you are going to do MPI programming we recommend getting hold of the document that refers to your implementation.

28.2 MPI Programming

MPI programming typically requires two components, a compiler and an MPI implementation.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_28, © Springer-Verlag London Limited 2012

There are also two ways of doing MPI programming

- a cluster or multiple systems running MPI
- a single system running MPI

In both cases an MPI installation will normally provide an MPI daemon or service that can then be called from an MPI program.

28.3 Compiler and Implementation Combination

A number of commercial companies provide a combined bundle including

- Cray
- IBM
- Intel
- PGI

The Cray and IBM offerings will most likely be for a cluster. Intel and PGI provide products for both clusters and single systems. You should check their sites for up to date information.

28.4 Individual Implementation

A low cost option is to get hold of an MPI implementation that works with your existing compiler, and install it yourself on your own system.

The Intel MPI product is available as a free download for evaluation purposes.

There are a number of free MPI implementations, and details are given below for two of them.

28.4.1 MPICH2

They are based at Argonne National Laboratory

`http://www.mcs.anl.gov/research/projects/mpich2/`

MPICH2 is distributed as source (with an open-source, freely available license). It has been tested on several platforms, including Linux (on IA32 and x86–64), Mac OS/X (PowerPC and Intel), Solaris (32- and 64-bit), and Windows.

28.4.2 Open MPI

They can be found at

`http://www.open-mpi.org/`

They develop Open MPI on Linux, OS X, Solaris (both 32 and 64 on all platforms) and Windows (Windows XP, Windows HPC Server 2003/2008 and also Windows 7 RC).

28.5 Compiler and MPI Combinations Used in the Book

The examples in this chapter have been tried out with a variety of compilers and implementations, including

- Intel compiler + mpich2, Windows
- Intel compiler + Intel MPI, Windows
- Gfortran + openmpi, SuSe Linux 11.x
- Cray compiler, Hector Service
- PGI compiler, Hector Service
- IBM compiler, Met Office Slovakia

We haven't tried out all of the examples with all of the compiler and MPI implementations.

28.6 The MPI Memory Model

MPI is characterised generally by distributed memory and

- All threads/processes have access to their own private memory only
- Data transfer and most synchronization has to be programmed explicitly
- All data is private
- Data is shared explicitly by exchanging buffers in MPI terminology

but in this chapter we will also show the use of MPI on one system.

28.7 Example 1 – Hello World

The first example is the classic hello world program. This example has been run on the following systems:

- Single system, Intel compiler and mpich2, Windows
- Single system, Intel compiler and Intel MPI, Windows
- Single system, gfortran and openmpi, SuSe linux
- Cray HECToR service, Edinburgh

Here is the program.

```
program ch2801
use mpi
implicit none
integer :: error_number
integer :: this_process_number
integer :: number_of_processes
   call MPI_INIT( error_number )
   call MPI_COMM_SIZE( MPI_COMM_WORLD,
number_of_processes , error_number )
   call MPI_COMM_RANK( MPI_COMM_WORLD,
this_process_number , error_number )
   print *, " Hello from process " ,
this_process_number , " of ", number_of_processes , "
processes!"
   call MPI_Finalize(error_number)
end program ch2801
```

Let us look at each statement in turn.

```
use mpi
```

With most modern MPI implementations we can make available the MPI setup with a use statement. Older implementations required an include file option.

```
   call MPI_INIT( error_number )
```

This must be the first MPI routine called. The Fortran binding only takes one argument, an integer variable that is used to return an error number. It sets up the MPI environment.

```
   call MPI_COMM_SIZE( MPI_COMM_WORLD,
number_of_processes , error_number )
```

is typically the second MPI routine called. All MPI communication is associated with a so called communicator that describes the communication context and an associated set of processes. In this simple example we use the default communicator, called MPI_COMM_WORLD. The number of processes available is returned via the second argument. This means that the above program is duplicated on each process, i.e. number_of_processes determines how many copies are running.

```
   call MPI_COMM_RANK( MPI_COMM_WORLD,
this_process_number , error_number )
```

The call above returns the process number for this process or copy of the program.

```
   print *, " Hello from process " ,
this_process_number , " of ", number_of_processes , "
processes!"
```

Each copy of the program will print out this message.

```
   call MPI_Finalize(error_number)
```

The call to MPI_Finalize is the last call to the MPI system we need to make.
Here is the output from the Intel compiler and Intel MPI option under Windows XP64.

```
mpiexec -n 8 ch2801
   Hello from process           0   of           8
processes!
   Hello from process           4   of           8
processes!
   Hello from process           1   of           8
processes!
   Hello from process           5   of           8
processes!
   Hello from process           7   of           8
processes!
   Hello from process           6   of           8
processes!
   Hello from process           3   of           8
processes!
   Hello from process           2   of           8
processes!
```

Notice that process numbering starts at 0. Note also that there is no particular order to the process numbers.

Here is the output from gfortran and openmpi on a SuSe 11.2 Linux box. This is the same system as the above, as it is dual boot.

```
  mpiexec -n 8 ch2801.out
     Hello from process           0   of
8    processes!
     Hello from process           1   of
8    processes!
     Hello from process           2   of
8    processes!
     Hello from process           3   of
8    processes!
     Hello from process           4   of
8    processes!
     Hello from process           5   of
8    processes!
     Hello from process           6   of
```

```
8    processes!
     Hello from process            7   of
8    processes!
```

Now the ordering is sequential.

Here is the output from the Cray HECToR service. This uses 64 processes. The job is submitted as a batch job, via a queueing mechanism. This is a common mechanism on larger multi user systems.

```
  Hello from process   3  of   64  processes!
  Hello from process   0  of   64  processes!
  Hello from process   1  of   64  processes!
  Hello from process   2  of   64  processes!
  Hello from process   61  of   64  processes!
  Hello from process   60  of   64  processes!
  Hello from process   63  of   64  processes!
  Hello from process   62  of   64  processes!
  Hello from process   56  of   64  processes!
  Hello from process   59  of   64  processes!
  Hello from process   57  of   64  processes!
  Hello from process   58  of   64  processes!
  Hello from process   40  of   64  processes!

...
...lines deleted
...
  Hello from process   33  of   64  processes!
  Hello from process   46  of   64  processes!
  Hello from process   45  of   64  processes!
  Hello from process   47  of   64  processes!
  Hello from process   44  of   64  processes!
  Hello from process   4  of   64  processes!
  Hello from process   7  of   64  processes!
  Hello from process   5  of   64  processes!
  Hello from process   6  of   64  processes!
```

The order appears to be pretty random!

28.8 Example 2 – Hello World Using Send and Receive

The following is a variation of the above. In the first example we had no communication between processes. Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are send and receive. Their use is shown in the example below. These are blocking send and receive operations. A blocking send does not return until the

message data and envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

In this example process 0 is the master process and this communicates with every other process or program.

```
program ch2802
use mpi
implicit none
integer :: error_number
integer :: this_process_number
integer :: number_of_processes
integer :: i
integer , dimension(MPI_STATUS_SIZE) :: status
   call MPI_INIT( error_number )
   call MPI_COMM_SIZE( MPI_COMM_WORLD,
number_of_processes , error_number )
   call MPI_COMM_RANK( MPI_COMM_WORLD,
this_process_number , error_number )
   if  ( this_process_number == 0 ) then
     print *, " Hello from process " ,
this_process_number , " of ", number_of_processes , "
processes."
     do i = 1 , number_of_processes - 1
       call MPI_RECV(this_process_number ,
1 , MPI_INTEGER , I , 1 , MPI_COMM_WORLD , status ,
error_number)
       print *, " Hello from process " ,
this_process_number , " of ", number_of_processes , "
processes."
     end do
   else
     call MPI_SEND(this_process_number , 1 , MPI_INTEGER
, 0 , 1 , MPI_COMM_WORLD , error_number)
   end if
   call MPI_Finalize(error_number)
end program ch2802
```

The calls to MPI_INIT, MPI_COMM_SIZE, MPI_COMM_RANK, MPI_Finalize are the same as in the first example. We have the additional code

- A test to see if we are process 0. If we are we then print out a message saying that we are process 0. We next loop from 1 to number_of_processes −1 and call MPI_RECV.
- If we are not process 0 we make a call to MPI_SEND – remember that the program executes on all processes.

Let us look at the calls to MPI_RECV and MPI_SEND in more depth. Here is an extract from the 2.2 specification describing MPI_RECV

- <type> BUF(*), initial address of receive buffer
- INTEGER COUNT , Number of elements in the receive buffer
- DATATYPE , data type of each receive buffer element
- SOURCE , rank of source
- TAG , message tag
- COMM , communicator
- STATUS(MPI_STATUS_SIZE),
- IERROR

The following shows the mapping between MPI data types and Fortran data types.

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

Our arguments to MPI_RECV are

- this_process_number - process 0 is doing the receiving
- 1 item
- MPI_INTEGER – an MPI_INTEGER variable
- I – receive from this process
- 1 – tag
- MPI_COMM_WORLD - the communicator
- status – an integer array of size MPI_STATUS_SIZE
- error_number

Here is an extract from the 2.2 specification regarding MPI_SEND

- <type> BUF(*) – initial address of send buffer
- INTEGER COUNT – number of elements in send buffer
- DATATYPE – data type of each send buffer element
- DEST – rank of destination
- TAG – message tag
- COMM – communicator
- IERROR – error number

The arguments to our MPI_SEND are

- this_process_number – send from this process
- 1
- MPI_INTEGER
- 0 – send to this process number

- 1
- MPI_COMM_WORLD – the communicator
- error_number

and as you can see the sends and receives are in matching pairs.

The output from this program will be similar to the previous example.

28.9 Example 3 – Serial Solution for pi Calculation

We choose numerical integration in this example. The following integral

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

is one way of calculating an approximation to π, and is a problem that is easy to parallelise. The integral can be approximated by

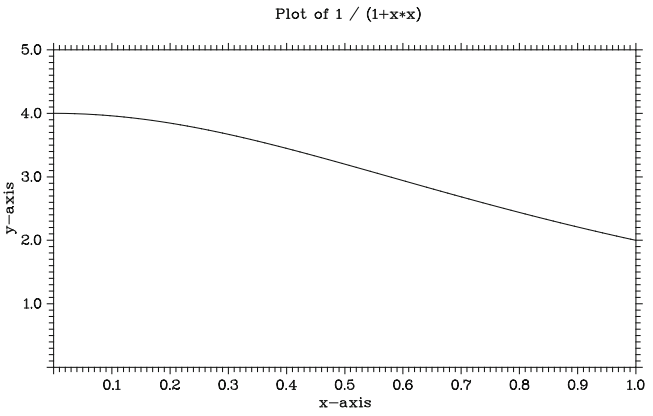
$$\frac{1}{n}\sum_1^n \frac{4}{1+\left(\frac{i-0.5}{n}\right)^2}$$

π to 50 digits is
3.14159265358979323846264338327950288419716939937510
according to Wikipedia

Another way of calculating π is using the formula $4\tan^{-1}(1)$, and in Fortran this is

`4.0*atan(1.0)`.

Consider the following plot of the above equation.



To do the evaluation numerically we divide the interval between 0 and 1 into n sub intervals. The higher the value of n the more accurate our value of π will be, or should be.

Here is a serial program to do this calculation. The program is in three main parts. These are

- precision module – to set the precision throughout the whole code.
- timing module – a timing module to enable us to time parts of the program. We will be using this module throughout the parallel examples to provide information about the performance of the algorithms.
- the program – that actually does the integration.

The first two modules are straightforward and we will only cover the integration solution in depth. We will be using this integration example in this chapter on MPI and the subsequent two on OpenMP and coarray Fortran.

```
module precision_module
   implicit none
   integer, parameter :: long =
selected_real_kind(15,307)
end module precision_module

module timing_module
   implicit none

   integer, dimension (8), private :: dt
   real, private :: h, m, s, ms, tt
   real, private :: last_tt

contains
   subroutine start_timing()
     implicit none

     call date_and_time(values=dt)
     print 100, dt(1:3), dt(5:8)
     100 format &
     (1x,i4,'/',i2,'/',i2,1x,i2,':',i2,':',i2,1x,i3)
     h = real(dt(5))
     m = real(dt(6))
     s = real(dt(7))
     ms = real(dt(8))
     last_tt = 60*(60*h+m) + s + ms/1000.0
   end subroutine start_timing

   subroutine print_date_and_time
     implicit none
```

```
    call date_and_time(values=dt)
    print 100, dt(1:3), dt(5:8)
    100 format &
    (1x,i4,'/',i2,'/',i2,1x,i2,':',i2,':',i2,1x,i3)
  end subroutine print_date_and_time

  subroutine print_hms
    implicit none

    call date_and_time(values=dt)
    print 100, dt(5:8)
    100 format (1x,i2,':',i2,':',i2,1x,i3)
  end subroutine print_hms

  subroutine print_ms
    implicit none

    call date_and_time(values=dt)
    h = real(dt(5))
    m = real(dt(6))
    s = real(dt(7))
    ms = real(dt(8))
    tt = 60*(60*h+m) + s + ms/1000.0
    print 100, tt
    100 format (1x,f14.3)
  end subroutine print_ms

  subroutine print_time_difference
    implicit none

    call date_and_time(values=dt)
    h = real(dt(5))
    m = real(dt(6))
    s = real(dt(7))
    ms = real(dt(8))
    tt = 60*(60*h+m) + s + ms/1000.0
    print 100, (tt-last_tt)
    100 format (1x,f14.3)
    last_tt = tt
  end subroutine print_time_difference

  real function time_difference()
    implicit none

    tt = 0.0
    call date_and_time(values=dt)
```

```
    h = real(dt(5))
    m = real(dt(6))
    s = real(dt(7))
    ms = real(dt(8))
    tt = 60*(60*h+m) + s + ms/1000.0
    time_difference = tt - last_tt
  end function time_difference
end module timing_module

program ch2803
  use precision_module
  use timing_module
  implicit none
  integer :: i, j
  integer :: n_intervals
  real (long) :: interval_width, x, total, pi
  real (long) :: fortran_internal_pi

  call start_timing()
  n_intervals = 10
  fortran_internal_pi = 4.0_long*atan(1.0_long)
  print *, ' fortran_internal_pi = ', &
  fortran_internal_pi
  print *, ' '
  do j = 1, 9
    interval_width = 1.0_long/n_intervals
    total = 0.0_long
    do i = 1, n_intervals
      x = interval_width*(real(i,long)-0.5_long)
      total = total + f(x)
    end do
    pi = interval_width*total
    print 20, n_intervals, time_difference()
    20 format (' N intervals = ',i12,' time = ',f8.3)
    print 30, pi, abs(pi-fortran_internal_pi)
    30 format (' pi = ',f20.16,/, &
    ' difference = ',f20.16)
    n_intervals = n_intervals*10
  end do
contains
  real (long) function f(x)
    implicit none
    real (long), intent (in) :: x

    f = 4.0_long/(1.0_long+x*x)
  end function f
end program ch2803
```

The first part of the code has the declarations for the variables we will be using. These are

```
integer :: n_intervals
real (long) :: interval_width, x, total, pi
real (long) :: fortran_internal_pi
```

We have an integer variable for the number of intervals we will be using. We have made this of default integer type, which will be 32 bit on most platforms, and will be up to 2,147,483,647.

We then have the following variables

- interval_width
- x – the variable we will be calculating numerically
- total – our total for the integration
- pi – our calculated value of pi
- fortran_internal_pi – we use a common way of defining this using the internal atan function.

We then call the start_timing routine to print out details of the start time.

We next set the number of intervals. We choose 10 as an initial value. We will be doing the calculation for a number of interval sizes.

We calculate pi using the atan intrinsic and print out its value. We will be using this value to determine the accuracy of our calculations.

We then have the loop that does the calculations for nine values of the interval size from 10 to 1,000,000,000.

We calculate the interval width at the start of each loop and reset the total to zero at the start of each loop.

The following

```
do i = 1, n_intervals
   x = interval_width*(real(i,long)-0.5_long)
   total = total + f(x)
end do
```

is the code that actually does the integration. We calculate x each time round the loop and then use this calculated value in our call to our function, summing up as we go along. We need to subtract a ½ as we need the midpoint of the interval for our value of x.

The loop finishes and we then calculate the value of pi and print out details of the number of intervals, the calculated value of pi and the difference between the internal value of pi and the calculated value.

We also print out timing information about this calculation. We then increment the number of intervals and repeat the above.

We need to know how long the serial version takes and how accurate our calculated value for pi is.

Here is output from this program on a couple of systems and compilers.

Compiler 1 - Intel 12.0.3, Windows Vista Home Premium x64

```
>ch2804
 2011/ 5/ 9 14:52:38 671
  fortran_internal_pi =     3.14159265358979

 N intervals =             10 time =       0.000
 pi =             3.1424259850010987
 difference =    0.0008333314113056
 N intervals =            100 time =       0.004
 pi =             3.1416009869231245
 difference =    0.0000083333333314
 N intervals =           1000 time =       0.004
 pi =             3.1415927369231267
 difference =    0.0000000833333336
 N intervals =          10000 time =       0.004
 pi =             3.1415926544231239
 difference =    0.0000000008333307
 N intervals =         100000 time =       0.004
 pi =             3.1415926535981167
 difference =    0.0000000000083236
 N intervals =        1000000 time =       0.012
 pi =             3.1415926535899033
 difference =    0.0000000000001101
 N intervals =       10000000 time =       0.051
 pi =             3.1415926535896861
 difference =    0.0000000000001070
 N intervals =      100000000 time =       0.445
 pi =             3.1415926535902168
 difference =    0.0000000000004237
 N intervals =     1000000000 time =       4.398
 pi =             3.1415926535897682
 difference =    0.0000000000000249
```

Compiler 2 - gfortran 4.4.1, SuSe Linux 11.2

```
> gfortran ch2804.f90 -O3 -funroll-loops -ffast-math
> ./a.out
  2011/ 5/ 9 15: 8:34 212
  fortran_internal_pi =     3.1415926535897931

 N intervals =             10 time =       0.000
 pi =             3.1424259850010987
 difference =    0.0008333314113056
```

```
 N intervals =            100 time =       0.000
 pi =             3.1416009869231254
 difference =    0.0000083333333323
 N intervals =           1000 time =       0.000
 pi =             3.1415927369231227
 difference =    0.0000000833333296
 N intervals =          10000 time =       0.000
 pi =             3.1415926544231341
 difference =    0.0000000008333410
 N intervals =         100000 time =       0.004
 pi =             3.1415926535981615
 difference =    0.0000000000083684
 N intervals =        1000000 time =       0.016
 pi =             3.1415926535897643
 difference =    0.0000000000000289
 N intervals =       10000000 time =       0.094
 pi =             3.1415926535897309
 difference =    0.0000000000000622
 N intervals =      100000000 time =       0.887
 pi =             3.1415926535904264
 difference =    0.0000000000006333
 N intervals =     1000000000 time =       8.777
 pi =             3.1415926535899708
 difference =    0.0000000000001776
```

Compiler 3 - Cray 4.7, Hector Service

```
> ./a.out
 2011/ 5/ 9 14:53:51 560
  fortran_internal_pi =   3.1415926535897931

 N intervals =              10 time =       0.004
 pi =             3.1424259850010987
 difference =    0.0008333314113056
 N intervals =             100 time =       0.004
 pi =             3.1416009869231245
 difference =    0.0000083333333314
 N intervals =            1000 time =       0.004
 pi =             3.1415927369231254
 difference =    0.0000000833333322
 N intervals =           10000 time =       0.004
 pi =             3.1415926544231318
 difference =    0.0000000008333387
 N intervals =          100000 time =       0.004
 pi =             3.1415926535981016
 difference =    0.0000000000083085
```

```
 N intervals =        1000000 time =       0.008
 pi =              3.1415926535899388
 difference =    0.0000000000001457
 N intervals =       10000000 time =       0.047
 pi =              3.1415926535899850
 difference =    0.0000000000001918
 N intervals =      100000000 time =       0.434
 pi =              3.1415926535900223
 difference =    0.0000000000002292
 N intervals =     1000000000 time =       4.297
 pi =              3.1415926535900072
 difference =    0.0000000000002141
```

The three sample runs provide us with information that we can use as a basis for an analysis of our parallel solution. We have information about the accuracy of the solution and timing details.

28.10 Example 4 – Parallel Solution for pi Calculation

This example is a parallel solution to the above problem using mpi. We only show the parallel program. The precision and timing modules are the same as in the previous example.

```
program ch2804
use precision_module
use timing_module
use mpi
implicit none
real (long) :: fortran_internal_pi
real (long) :: partial_pi
real (long) :: total_pi
real (long) :: width
real (long) :: partial_sum
real (long) :: x
integer :: n
integer :: this_process
integer :: n_processes
integer :: i
integer :: j
integer :: error_number

   call mpi_init(error_number)
   call mpi_comm_size(mpi_comm_world , &
```

```
   n_processes, error_number)
   call mpi_comm_rank(mpi_comm_world , &
   this_process ,error_number)
   n = 100000
   fortran_internal_pi = 4.0_long*atan(1.0_long)
   if (this_process==0) then
     call start_timing()
     print *, ' fortran_internal_pi = ',
fortran_internal_pi
   end if

   do j = 1, 5
     width = 1.0_long/n
     partial_sum = 0.0_long
     do i = this_process + 1, n, n_processes
       x = width*(real(i,long)-0.5_long)
       partial_sum = partial_sum + f(x)
     end do
     partial_pi = width*partial_sum
     call mpi_reduce(partial_pi ,total_pi ,&
      1 ,mpi_double_precision ,mpi_sum ,0, &
         mpi_comm_world ,error_number)
     if (this_process==0) then
       print 20, n, time_difference()
       20 format (' N intervals = ',i12, &
       ' time =   ',f8.3)
       print 30, total_pi , &
       abs(total_pi-fortran_internal_pi)
       30 format (' pi =            ',f20.16,/, &
       ' difference =  ',f20.16)
     end if
     n = n*10
   end do
   call mpi_finalize(error_number)

contains
   real (long) function f(x)
     implicit none
     real (long), intent (in) :: x

     f = 4.0_long/(1.0_long+x*x)
   end function f
end program ch2804
```

The first difference is the

```
use mpi
```

statement. This makes available the mpi functionality. We next have several variable declarations.

```
real (long) :: fortran_internal_pi
real (long) :: partial_pi
real (long) :: total_pi
real (long) :: width
real (long) :: partial_sum
real (long) :: x
integer :: n
integer :: this process
integer :: n_processes
integer :: i
integer :: j
integer :: error_number
```

The variables partial_pi, total_pi and partial_sum are required by our parallel algorithm. The variable n is the number of intervals and we start this at 100,000 rather than 10 as we have seen from the serial solution that there are quite large differences between the internal value of pi and the calculated value below 100,000.

The variables this _process, n_processes and error_number are required for the mpi solution.

The real work is done in the following do loop.

```
do i = this_process + 1, n, n_processes
  x = width*(real(i,long)-0.5_long)
  partial_sum = partial_sum + f(x)
end do
```

The key is to split up the work of the calculation between the processes we have available. The following shows how the work will be split up for n = 10 and with the number of processes ranging from 1 to 8.

```
n_processes=1 do i=1,n,1   1,2,3,4,5,6,7,8,9,10
n_processes=2 do i=1,n,2   1,3,5,7,9
              do i=2,n,2   2,4,6,8,10
n_processes=4 do i=1,n,4   1,5,9
              do i=2,n,4   2,6,10
              do i=3,n,5   3,7
              do i=4,n,4   4,8
n_processes=8 do i=1,n,8   1,9
              do i=2,n,8   2,10
              do i=3,n,8   3
```

```
do i=4,n,8   4
do i=5,n,8   5
do i=6,n,8   6
do i=7,n,8   7
do i=8,n,8   8
```

The above also shows how the algorithm balances the load of the computation across the processes.

Each process has its own partial_sum and partial_pi. We then use the call to the MPI subroutine mpi_reduce to calculate the total value of pi from the partial values of pi. Here is the MPI description of the mpi_reduce routine

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
IN sendbuf address of send buffer (choice)
OUT recvbuf address of receive buffer (choice, significant only at root)
IN count number of elements in send buffer (non-negative integer)
IN datatype data type of elements of send buffer (handle)
IN op reduce operation (handle)
IN root rank of root process (integer)
IN comm communicator (handle)

and

partial_pi is our send buffer
total_pi is our receive buffer
1 – the number of elements
mpi_double_precision – the type of the elements
mpi_sum – the reduction operation
0 – the root process
mpi_comm_world – the communicator
error_number – the error number

We then control the printing from process 0.

Here is sample output from the Intel compiler for 1, 4, 8 and 32 processes. Results are similar from gfortran. We can control how many processes are available from the command line. We will look at the Cray compiler and the Hector service later.

One process

```
mpiexec -n 1 total_ch2804
 2011/ 5/ 9 16: 8:27 243
  fortran_internal_pi =     3.14159265358979
 N intervals =         100000 time =      0.000
 pi =             3.1415926535981016
 difference =    0.0000000000083085
 N intervals =        1000000 time =      0.004
 pi =             3.1415926535899388
 difference =    0.0000000000001457
```

```
 N intervals =       10000000 time =       0.043
 pi =              3.1415926535899850
 difference =    0.0000000000001918
 N intervals =      100000000 time =       0.441
 pi =              3.1415926535900223
 difference =    0.0000000000002292
 N intervals =     1000000000 time =       4.406
 pi =              3.1415926535900072
 difference =    0.0000000000002141
```

4 processes

```
mpiexec -n 4 total_ch2804
 2011/ 5/ 9 16: 8:49 555
  fortran_internal_pi =     3.14159265358979
 N intervals =          100000 time =       0.000
 pi =              3.1415926535981349
 difference =    0.0000000000083418
 N intervals =        1000000 time =       0.004
 pi =              3.1415926535898899
 difference =    0.0000000000000968
 N intervals =       10000000 time =       0.012
 pi =              3.1415926535898069
 difference =    0.0000000000000138
 N intervals =      100000000 time =       0.109
 pi =              3.1415926535896137
 difference =    0.0000000000001794
 N intervals =     1000000000 time =       1.098
 pi =              3.1415926535898278
 difference =    0.0000000000000346
```

8 processes

```
mpiexec -n 8 total_ch2804
 2011/ 5/ 9 16: 9: 1 354
  fortran_internal_pi =     3.14159265358979
 N intervals =          100000 time =       0.000
 pi =              3.1415926535981260
 difference =    0.0000000000083329
 N intervals =        1000000 time =       0.000
 pi =              3.1415926535898802
 difference =    0.0000000000000870
 N intervals =       10000000 time =       0.012
 pi =              3.1415926535897936
 difference =    0.0000000000000004
 N intervals =      100000000 time =       0.109
 pi =              3.1415926535897745
```

```
 difference =    0.0000000000000187
 N intervals =    1000000000 time =       1.102
 pi =             3.1415926535898446
 difference =    0.0000000000000515
```

32 processes

```
mpiexec -n 32 total_ch2804
 2011/ 5/ 9 16: 9:16 345
  fortran_internal_pi =      3.14159265358979
 N intervals =         100000 time =       0.715
 pi =             3.1415926535981265
 difference =    0.0000000000083333
 N intervals =        1000000 time =       0.715
 pi =             3.1415926535898753
 difference =    0.0000000000000822
 N intervals =       10000000 time =       0.719
 pi =             3.1415926535897953
 difference =    0.0000000000000022
 N intervals =     100000000 time =       0.742
 pi =             3.1415926535897936
 difference =    0.0000000000000004
 N intervals =    1000000000 time =       1.125
 pi =             3.1415926535898087
 difference =    0.0000000000000155
```

The system that the above output is from has an Intel Core i7 920 processor. This processor has four cores and each core is hyper threaded. We get a nearly linear speed up to four processes, which shows how good the parallel solution is. Note that the time value is not the total time taken by all processes, but rather the effective running time of the program. If we are sat in front of the pc the program would complete in about a quarter of the time of the serial version. The numerical results are similar to the serial solution. The eight and thirty two process versions have similar times to the four process version.

Here is the output from the Cray at the Hector service. This is for 64 processes running on 16 nodes. Each compute node contains two AMD 2.1 GHz 12 core processors.

```
 2011/ 5/ 9 17:29:56 509
  fortran_internal_pi =   3.1415926535897931
 N intervals =         100000 time =       0.004
 pi =             3.1415926535981265
 difference =    0.0000000000083333
 N intervals =        1000000 time =       0.004
 pi =             3.1415926535898762
 difference =    0.0000000000000830
 N intervals =       10000000 time =       0.004
```

```
 pi =              3.1415926535897971
 difference =    0.0000000000000040
 N intervals =     100000000 time =       0.020
 pi =              3.1415926535897931
 difference =    0.0000000000000000
 N intervals =    1000000000 time =       0.180
 pi =              3.1415926535897865
 difference =    0.0000000000000067
```

As can be seen this represents a major time reduction over the serial version from 4.297 to 0.180 s – a factor of approximately 24.

Here is the output from 96 processes on 4 nodes.

```
 2011/ 5/ 9 18:26:32 91
  fortran_internal_pi =   3.1415926535897931
 N intervals =         100000 time =       0.000
 pi =              3.1415926535981256
 difference =    0.0000000000083324
 N intervals =        1000000 time =       0.000
 pi =              3.1415926535898757
 difference =    0.0000000000000826
 N intervals =       10000000 time =       0.000
 pi =              3.1415926535897949
 difference =    0.0000000000000018
 N intervals =     100000000 time =       0.016
 pi =              3.1415926535897905
 difference =    0.0000000000000027
 N intervals =    1000000000 time =       0.125
 pi =              3.1415926535897949
 difference =    0.0000000000000018
```

Again we have a considerable speed up, a factor of approximately 35.

28.11 Example 5 – Work Sharing Between Processes

This example looks at one way of splitting work up between processes. We use the process number of determine which process does which work.

```
program ch2805
use mpi
implicit none
integer :: error_number
integer :: this_process_number
integer :: number_of_processes
integer, dimension (mpi_status_size) :: status
```

```
integer, allocatable, dimension (:) :: x
integer :: n
integer, parameter :: factor = 5
integer :: i, j, k
integer :: start
integer :: end
integer :: recv_start

  call mpi_init(error_number)
  call mpi_comm_size(mpi_comm_world, &
    number_of_processes,error_number)
  call mpi_comm_rank(mpi_comm_world,&
    this_process_number,error_number)
  n = number_of_processes*factor
  allocate (x(1:n))
  x = 0
  start = (factor*this_process_number) + 1
  end = factor*(this_process_number+1)
  print 10,this_process_number,start,end
  10 format(' Process number = ',i3,' start ', &
    i3,' end ',i3)
  do i = start, end
    x(i) = i*factor
  end do
  do i = 1, n
    print 20 , this_process_number, i, x(i)
    20 format(1x,i4, ' i ',i4,' x(i) ',i4)
  end do
  if (this_process_number==0) then
    do i = 1, number_of_processes - 1
      recv_start = (factor*i)+ 1
      call mpi_recv(x(recv_start),&
        factor,mpi_integer,i,1,mpi_comm_world, &
        status ,error_number)
    end do
  else
    call mpi_send(x(start),factor, &
      mpi_integer,0,1,mpi_comm_world,error_number)
  end if
  if (this_process_number==0) then
    do i = 1, n
      print 30, i, factor, x(i)
      30 format (1x,i4,' * ',i2,' = ',i5)
    end do
  end if
  call mpi_finalize(error_number)
end program ch2805
```

What we are going to do is allocate an array based on the number of processes and then split the (simple) work on the array up between the processes. We will calculate array indices from the process numbers.

```
n = number_of_processes*factor
```

This statement calculates the array size based on the number of processes and a constant factor.

```
allocate (x(1:n))
```

This statement allocates the array.

```
x = 0
```

This statement initialises the whole array to zero. The following statements define the start and end points for the array processing for each process.

```
start = (factor*this_process_number) + 1
end = factor*(this_process_number+1)
```

and partition the work up between the processes. Each process will have its own start and end values. The following do loop does the work:

```
do i = start, end
   x(i) = i*factor
end do
```

and all we are doing as this is filling sections of the array up with data based in process numbers.

The following

```
if (this_process_number==0) then
   do i = 1, number_of_processes - 1
      recv_start = (factor*i) + 1
      call mpi_recv(x(recv_start), &
         factor,mpi_integer,i,1,mpi_comm_world,&
         status ,error_number)
   end do
else
   call mpi_send(x(start),factor, &
      mpi_integer,0,1,mpi_comm_world,error_number)
end if
```

uses sends and receives to transfer the updated array sections back to process zero. We are using recv_start to specify the starting point for the array transfer, and x(start) is the starting point for the transfer from the x array to process zero.

Here is sample output from the program when the number of processes is three.

```
mpiexec -n 3 ch2805
 Process number =    2 start  11 end  15
 Process number =    1 start   6 end  10
     1 I     1 x(i)     0
     1 I     2 x(i)     0
     1 I     3 x(i)     0
     1 I     4 x(i)     0
     1 I     5 x(i)     0
     1 I     6 x(i)    30
     1 I     7 x(i)    35
     1 I     8 x(i)    40
     1 I     9 x(i)    45
     1 I    10 x(i)    50
     1 I    11 x(i)     0
     1 I    12 x(i)     0
 Process number =    0 start   1 end   5
     0 I     1 x(i)     5
     0 I     2 x(i)    10
     0 I     3 x(i)    15
     0 I     4 x(i)    20
     0 I     5 x(i)    25
     0 I     6 x(i)     0
     0 I     7 x(i)     0
     0 I     8 x(i)     0
     0 I     9 x(i)     0
     0 I    10 x(i)     0
     0 I    11 x(i)     0
     0 I    12 x(i)     0
     0 I    13 x(i)     0
     0 I    14 x(i)     0
     0 I    15 x(i)     0
     1 *  5 =      5
     2 *  5 =     10
     3 *  5 =     15
     4 *  5 =     20
     5 *  5 =     25
     6 *  5 =     30
     7 *  5 =     35
     2 I    1 x(i)      0
     2 I    2 x(i)      0
```

```
 2 I    3 x(i)       0
 2 I    4 x(i)       0
 2 I    5 x(i)       0
 2 I    6 x(i)       0
 2 I    7 x(i)       0
 2 I    8 x(i)       0
 2 I    9 x(i)       0
 2 I   10 x(i)       0
 2 I   11 x(i)      55
 2 I   12 x(i)      60
 2 I   13 x(i)      65
 2 I   14 x(i)      70
 2 I   15 x(i)      75
 1 I   13 x(i)       0
 1 I   14 x(i)       0
 1 I   15 x(i)       0
 8 *   5 =      40
 9 *   5 =      45
10 *   5 =      50
11 *   5 =      55
12 *   5 =      60
13 *   5 =      65
14 *   5 =      70
15 *   5 =      75
```

So with three processes we have an array of size 15, and the work that each process does is

- Process number = 0 start 1 end 5
- Process number = 1 start 6 end 10
- Process number = 2 start 11 end 15

and each process works on its own section of the array. At the end we use the sends and receives to make sure that the x array on process zero now has all of the updated values.
This code achieves load balancing across the processes.

28.12 Summary

The programs in this chapter provide an introduction to the use of MPI to achieve parallel programs in Fortran. We have also seen some of the timing benefits of parallel programming with MPI.

28.13 Problem

1. Compile and run the programs with your compiler and implementation of MPI. You should get similar results.

Chapter 29
OpenMP

The best way to have a good idea is to have a lot of ideas.

Linus Pauling

Aim

The aims of this chapter is to provide a short introduction to OpenMP programming in Fortran.

29.1 Introduction

The main site is

`http://openmp.org/wp/`

and this site has details about the various specifications

`http://openmp.org/wp/openmp-specifications/`

We recommend downloading the documentation if you are going to do OpenMP programming. You should visit

`http://openmp.org/wp/openmp-compilers/`

to see an up to date list of what compilers support the OpenMP specification, and at what level.

The OpenMP site has a range of resources available, check out

`http://openmp.org/wp/resources`

for more information.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_29, © Springer-Verlag London Limited 2012

We've run the examples in this chapter with one or more of the following compilers

- Cray
- Gfortran
- Intel

29.2 OpenMP Memory Model

OpenMP is a shared memory programming model. It has several features including

- All threads have access to the same shared memory
- Data can be shared or private
- Data transfer is transparent to the programmer
- Synchronization takes place and is generally implicit

We will look at a small number of examples to highlight some of the key features. We provide a brief coverage of some of the OpenMP glossary to provide a basic background to OpenMP.

- Threading Concepts
  - Thread – An execution entity with a stack and associated static memory, called threadprivate memory.
  - OpenMP thread – A thread that is managed by the OpenMP runtime system.
  - Thread-safe routine – A routine that performs the intended function even when executed concurrently (by more than one thread).
- OpenMP language terminology
  - Structured block – For Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.
  - Loop directive – An OpenMP executable directive whose associated user code must be a loop that is a structured block. For Fortran, only the do directive and the optional end do directive are loop directives.
  - Master thread – The thread that encounters a parallel construct, creates a team, generates a set of tasks, then executes one of those tasks as thread number 0.
  - Worksharing construct – A construct that defines units of work, each of which is executed exactly once by one of the threads in the team executing the construct. For Fortran, worksharing constructs are do, sections, single and workshare.
  - Barrier – A point in the execution of a program encountered by a team of threads, beyond which no thread in the team may execute until all threads in the team have reached the barrier and all explicit tasks generated by the team have executed to completion.
- Data Terminology
  - Variable – A named data object, whose value can be defined and re defined during the execution of a program. Only an object that is not part of another object is considered a variable. For example, array elements, structure components, array sections and substrings are not considered variables.

  - Private variable – With respect to a given set of task regions that bind to the same parallel region, a variable whose name provides access to a different block of storage for each task region.
  - Shared variable – With respect to a given set of task regions that bind to the same parallel region, a variable whose name provides access to the same block of storage for each task region.

- Execution Model
  - The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library).

The above coverage should be enough to get started with OpenMP and understand the examples that follow.

29.3 Example 1 – Hello World

This is the classic hello world program.

```
program ch2901
use omp_lib
implicit none
integer :: nthreads
integer :: thread_number
integer :: i

  nthreads = omp_get_max_threads()
  print *, ' Number of threads = ', nthreads
!$omp parallel do
  do i = 1, nthreads
    print *, ' Hello from thread ',&
        omp_get_thread_num()
  end do
!$omp end parallel do
end program ch2901
```

Let us go through the program one statement at a time.

```
use omp_lib
```

This use statement makes available the OpenMP environment. OpenMP statements are treated as comments without this statement.

```
  nthreads = omp_get_max_threads()
  print *, ' Number of  threads = ', nthreads
```

The first statement sets the variable nthread to the value returned by the OpenMP function omp_get_max_threads(). We then print out this value.

```
!$omp parallel do
```

OpenMP directives in Fortran start with the comment character (!), followed by a $ symbol and the characters omp. We use this form as it is works with both free format and fixed format Fortran source code.

The parallel do words indicate that the code that follows is a parallel region construct. In this case a do loop. Here is a small table listing some of the OpenMP directives.

Parallel region construct
!$omp parallel [clause] structured block !$omp end parallel
Work sharing constructs
!$omp do [clause] … do loop !$omp end parallel
!$omp sections [clause] … [!$omp section structured block] … !$omp end sections [nowait]
!$omp single [clause] structured block !$omp end single [nowait]
Combined parallel work sharing constructs
!$omp parallel do [clause] structured block !$omp end parallel do
!$omp parallel sections [clause] … [!$omp section structured block] … !$omp end parallel sections
Synchronisation constructs
!$omp master structured block ! $omp end master
!$omp critical [(name)] structured block !$omp end critical [(name)]
!$omp barrier
$omp atomic expression list
!$omp flush

!$omp ordered structured block !$omp end ordered
Data environment
!$omp threadprivate (/c1/,/c2/)

We next have the parallel do.

```
  do i = 1, nthreads
    print *, ' Hello from thread ',
omp_get_thread_num()
  end do
```

This loop prints out a message from each thread showing the thread number.

```
!$omp end parallel do
```

This marks the end of the OpenMP parallel loop.

So at the start of the loop the OpenMP run time system does a fork and creates multiple threads. At the end of the loop we have a join operation and we are back to one thread of execution.

Here is the output from the Intel compiler on an Intel i7 system.

```
  Number of threads =                8
  Hello from thread                  0
  Hello from thread                  4
  Hello from thread                  2
  Hello from thread                  3
  Hello from thread                  1
  Hello from thread                  7
  Hello from thread                  6
  Hello from thread                  5
```

These Intel systems have four real cores and each core supports hyper threading in Intel terminology. So the OpenMP system sees eight threads.

Here is the output from the gfortran compiler on the same system.

```
 Number of threads  =               8
  Hello from thread                 1
  Hello from thread                 3
  Hello from thread                 2
  Hello from thread                 4
  Hello from thread                 5
  Hello from thread                 6
  Hello from thread                 0
  Hello from thread                 7
```

The output is very similar, as one would expect.

29.4 Example 2 – Hello World Using Default Variable Data Scoping

This is a simple variation on the first example. At first sight it appears to be identical in effect to example one

```
program ch2902
use omp_lib
implicit none
integer :: nthreads
integer :: thread_number
integer :: i

  nthreads = omp_get_max_threads()
  print *, ' Number of threads = ', nthreads
!$omp parallel do
  do i = 1, nthreads
    thread_number = omp_get_thread_num()
    print *, ' Hello from thread ', thread_number
  end do
!$omp end parallel do
end program ch2902
```

However we have introduced a variable thread_number and are using the OpenMP default data scoping rules, i.e. we have said nothing. Here is the output from the Intel compiler.

```
ch2902
   Number of threads  =                  8
   Hello from thread                    4
   Hello from thread                    5
   Hello from thread                    0
   Hello from thread                    1
   Hello from thread                    2
   Hello from thread                    3
   Hello from thread                    7
   Hello from thread                    6
```

We appear to have a working program. Here is the output from the gfortran compiler.

```
$ ./a.exe
  Number of threads  =                  8
  Hello from thread                    6
  Hello from thread                    7
  Hello from thread                    7
  Hello from thread                    7
  Hello from thread                    7
```

```
  Hello from thread                    7
  Hello from thread                    7
  Hello from thread                    7
```

Now something appears to be not quite right! The default variable scoping rules mean that the variable thread_number is available to all threads – in OpenMP terminology it is shared. The opposite of shared is private and each thread has their own copy. Example 3 corrects this problem.

29.5 Example 3 – Hello World with Private thread_number Variable

```
program ch2903
use omp_lib
implicit none
integer :: nthreads
integer :: thread_number
integer :: i

   nthreads = omp_get_max_threads()
   print *, ' Number of threads = ', nthreads
!$omp parallel do private(thread_number)
   do i = 1, nthreads
      thread_number = omp_get_thread_num()
      print *, ' Hello from thread ', thread_number
   end do
!$omp end parallel do
end program ch2903
```

Here is the output from the gfortran compiler.

```
$ ./a.exe
  Number of threads =                  8
  Hello from thread                    2
  Hello from thread                    1
  Hello from thread                    4
  Hello from thread                    3
  Hello from thread                    0
  Hello from thread                    6
  Hello from thread                    5
  Hello from thread                    7
```

Care must be taken with variables in OpenMP to ensure they have the correct data scoping state.

29.6 Example 4 – Parallel Solution for pi Calculation

This is an OpenMP parallel implementation of the integration problem (example three) from the previous chapter. You should compare it with the MPI solution – example four in the last chapter.

```
program ch2904
use precision_module
use timing_module
use omp_lib
implicit none
real (long) :: fortran_internal_pi
real (long) :: partial_pi
real (long) :: openmp_pi
real (long) :: width
real (long) :: x
integer :: nthreads
integer :: i
integer :: j
integer :: k
integer :: n
    nthreads = omp_get_max_threads()
    fortran_internal_pi = 4.0_long*atan(1.0_long)
    print *, ' Maximum number of threads is ', nthreads
    k=1
    do
        call start_timing()
        n = 100000
        call omp_set_num_threads(k)
        print *, ' Number of threads = ', k
        do j = 1, 5
            width = 1.0_long/n
            partial_pi = 0.0_long
            !$OMP parallel do private(x) shared(width)
reduction(+:partial_pi)
            do i = 1, n
               x = width*(real(i,long)-0.5_long)
               partial_pi = partial_pi + f (x)
            end do
            !$omp end parallel do
            openmp_pi = width*partial_pi
            print 20, n, time_difference()
            20 format (' N intervals = ',i12, &
              ' time =',f8.3)
            print 30, openmp_pi , &
              abs(openmp_pi-fortran_internal_pi)
```

```
            30 format (' openmp_pi  = ' , &
              f20.16,/,'difference = ',f20.16)
            n = n*10
          end do
          k=k*2
          if (k>nthreads) exit
      end do
contains
real (long) function f(x)
implicit none
real (long), intent (in) :: x
  f = 4.0_long/(1.0_long+x*x)
end function f
end program ch2904
```

Here is the output from the Intel compiler.

```
  Maximum number of threads is                    8
 2011/ 6/10 11:55:24 146
  Number of threads =                 1
 N intervals =          100000 time =     0.004
 openmp_pi =     3.1415926535981167
difference =     0.0000000000083236
 N intervals =         1000000 time =     0.012
 openmp_pi =     3.1415926535899033
difference =     0.0000000000001101
 N intervals =        10000000 time =     0.051
 openmp_pi =     3.1415926535896861
difference =     0.0000000000001070
 N intervals =       100000000 time =     0.449
 openmp_pi =     3.1415926535902168
difference =     0.0000000000004237
 N intervals =      1000000000 time =     4.398
 openmp_pi =     3.1415926535897682
difference =     0.0000000000000249
 2011/ 6/10 11:55:28 545
  Number of threads =                 2
 N intervals =          100000 time =     0.000
 openmp_pi =     3.1415926535981260
difference =     0.0000000000083329
 N intervals =         1000000 time =     0.000
 openmp_pi =     3.1415926535898624
difference =     0.0000000000000693
 N intervals =        10000000 time =     0.020
 openmp_pi =     3.1415926535897829
difference =     0.0000000000000102
 N intervals =       100000000 time =     0.219
```

```
 openmp_pi =     3.1415926535898926
difference =     0.0000000000000995
 N intervals =      1000000000 time =     2.195
 openmp_pi =     3.1415926535897380
difference =     0.0000000000000551
 2011/ 6/10 11:55:30 744
  Number of threads =                4
 N intervals =          100000 time =     0.004
 openmp_pi =     3.1415926535981287
difference =     0.0000000000083356
 N intervals =         1000000 time =     0.004
 openmp_pi =     3.1415926535898726
difference =     0.0000000000000795
 N intervals =        10000000 time =     0.027
 openmp_pi =     3.1415926535898153
difference =     0.0000000000000222
 N intervals =       100000000 time =     0.137
 openmp_pi =     3.1415926535898038
difference =     0.0000000000000107
 N intervals =      1000000000 time =     1.781
 openmp_pi =     3.1415926535898544
difference =     0.0000000000000613
 2011/ 6/10 11:55:32 524
  Number of threads =                8
 N intervals =          100000 time =     0.000
 openmp_pi =     3.1415926535981278
difference =     0.0000000000083347
 N intervals =         1000000 time =     0.004
 openmp_pi =     3.1415926535898784
difference =     0.0000000000000853
 N intervals =        10000000 time =     0.016
 openmp_pi =     3.1415926535897962
difference =     0.0000000000000031
 N intervals =       100000000 time =     0.113
 openmp_pi =     3.1415926535898162
difference =     0.0000000000000231
 N intervals =      1000000000 time =     1.137
 openmp_pi =     3.1415926535898824
difference =     0.0000000000000893
```

We have similar timing improvements to the MPI solutions.

29.7 Summary

This chapter briefly introduced the essentials of OpenMP programming. We have also seen the timing benefits that OpenMP programming can offer in the solution of the same problem

29.8 Problem

1. Compile and run the examples in this chapter with your compiler and compare the results.

Chapter 30
Coarray Fortran

Science is a wonderful thing if one does not have to earn one's living at it.

Einstein

Aim

The aims of this chapter is to provide a short introduction to coarray programming in Fortran.

30.1 Introduction

Coarrays were the major component of the Fortran 2008 standard. As stated earlier they are based on a single program multiple data model. Coarrays are a simple parallel programming extension to Fortran. They are effectively variables that can be shared across multiple instances of the same program or images in Fortran terminology.

Coarray variables look like conventional Fortran arrays, except that they use [] brackets instead of () brackets. In the simple declaration below

```
character(len=20) :: name[*]='*****'
```

we declare `name` to be a `character` coarray and the * in the [] brackets means that the bounds of the coarray are calculated at run time, rather than compile time.

```
read  *, name
```

is a reference to the coarray on the current image.
We can then use the following statement

```
name[i] = name
```

to broadcast the value read in to each of the other images.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_30, © Springer-Verlag London Limited 2012

Note the Fortran coarray syntax here. We use the [] brackets to reference the coarray variable on other images and the omission of the [] brackets is a reference to the coarray variable on the current image.

30.2 Coarray Terminology

The following is taken from the standard and covers some of the basic coarray terminology.

- CODIMENSION attribute – The CODIMENSION attribute species that an entity is a coarray. The coarray-spec specifies its corank or corank and cobounds.
- Allocatable coarray – A coarray with the ALLOCATABLE attribute has a specified corank, but its cobounds are determined by allocation or argument association.
- Explicit-coshape coarray – An explicit-coshape coarray is a named coarray that has its corank and cobounds declared by an explicit-coshape-spec.
- Coindexed named objects – A coindexed-named-object is a named scalar coarray variable followed by an image selector.
- Image selectors – An image selector determines the image index for a coindexed object.
- Image execution control and image control statements – The execution sequence on each image is specified in 2.3.5 of the standard.
- Execution of an image control statement divides the execution sequence on an image into segments. Each of the following is an image control statement:
  - SYNC ALL statement;
  - SYNC IMAGES statement;
  - SYNC MEMORY statement;
  - ALLOCATE or DEALLOCATE statement that has a coarray allocate-object;
  - CRITICAL or END CRITICAL;
  - LOCK or UNLOCK statement;
  - Any statement that completes execution of a block or procedure and which results in the implicit deallocation of a coarray;
  - STOP statement;
  - END statement of a main program.
- Coarray – A coarray is a data entity that has nonzero corank; it can be directly referenced or defined by any image. It may be a scalar or an array.
- Coarray dummy variables – If the dummy argument is a coarray, the corresponding actual argument shall be a coarray and shall have the VOLATILE attribute if and only if the dummy argument has the VOLATILE at tribute.
- Coarray intrinsics
  - image_index – convert a cosubscript to an image index
  - lcobound – cobounds of a coarray

- num_images – the number of images
- this_image – image index or cosubscripts
- ucobound – cobounds of a coarray

Let us look now at some simple examples.

30.3 Example 1 – Hello World

The first is the classic Hello world.

```
program ch3001
implicit none
  print *,' Hello world from image ', this_image()
end program ch3001
```

Here is the output from the Intel compiler.

```
>ch3001
   Hello world from image             5
   Hello world from image             3
   Hello world from image             4
   Hello world from image             8
   Hello world from image             1
   Hello world from image             6
   Hello world from image             2
   Hello world from image             7
```

The output is obviously very similar to the corresponding MPI and OpenMP versions.

30.4 Example 2 – Broadcasting Data

Here is a simple program that broadcasts data from one image to the rest. This is a common requirement in parallel programming.

```
program ch3002
implicit none
integer :: i
character(len=20) :: name[*]='*****'
  print  10,name,this_image()
  10 format(1x,' Hello ', a20,' from image ',i3)
  if (this_image() == 1) then
    print *,' Type in your name'
```

```
    read *, name
    do i = 2, num_images()
      name[i] = name
    end do
  end if
  sync all
  print  10,name,this_image()
end program ch3002
```

Here is the output from the Intel compiler.

```
>ch3002
   Hello  *****                          from image    1
   Hello  *****                          from image    3
   Hello  *****                          from image    5
   Hello  *****                          from image    7
   Hello  *****                          from image    2
   Hello  *****                          from image    4
   Hello  *****                          from image    8
   Type in your name
   Hello  *****                          from image    6
Jane
   Hello Jane                            from image    4
   Hello Jane                            from image    8
   Hello Jane                            from image    2
   Hello Jane                            from image    6
   Hello Jane                            from image    7
   Hello Jane                            from image    3
   Hello Jane                            from image    5
   Hello Jane                            from image    1
```

Again no particular ordering of the image numbers.

30.5 Example 3 – Parallel Solution for Pi Calculation

```
program ch3003
use precision_module
use timing_module
implicit none
real (long) :: fortran_internal_pi
real (long) :: partial_pi
real (long) :: coarray_pi
real (long) :: width
real (long) :: total_sum
```

```
real (long) :: x
real (long) , codimension[*] :: partial_sum
integer      :: n_intervals
integer      :: I
integer      :: j
integer      :: current_image
integer      :: n_images
  fortran_internal_pi = 4.0_long*atan(1.0_long)
  n_images=num_images()
  current_image=this_image()
  if (current_image == 1) then
    print *,' Number of images = ',n_images
  end if
  n_intervals=100000
  do j=1,5
    if (current_image == 1) then
      call start_timing()
    end if
    width = 1.0_long/real(n_intervals,long)
    total_sum=0.0_long
    partial_sum= 0.0_long
    do i=current_image,n_intervals,n_images
      x = (real(i,long) - 0.5_long)*width
      partial_sum = partial_sum + f(x)
    end do
    partial_sum=partial_sum*width
    sync all
    if (current_image==1) then
      do i=1,n_images
        total_sum=total_sum+partial_sum[i]
      end do
      coarray_pi = total_sum
      print 20, n_intervals, time_difference()
      20 format (' n intervals = ',i12,' time =',f8.3)
      print 30, coarray_pi , &
        abs(coarray_pi-fortran_internal_pi)
      30 format (' pi = ',f20.16,/, &
        ' difference = ',f20.16)
    end if
    n_intervals=n_intervals*10
    sync all
  end do

contains

real (long) function f(x)
implicit none
```

```
real (long), intent (in) :: x
  f = 4.0_long/(1.0_long+x*x)
end function f

end program ch3003
```

Here is the output from the Intel compiler.

```
 Number of images =              8
2011/ 6/10 13:40:48 479
n intervals =        100000 time =    0.004
pi =   3.1415926535981260
difference =   0.0000000000083329
2011/ 6/10 13:40:48 486
n intervals =       1000000 time =    0.004
pi =    3.1415926535898802
difference =   0.0000000000000870
2011/ 6/10 13:40:48 490
n intervals =      10000000 time =    0.012
pi =    3.1415926535897936
difference =   0.0000000000000004
2011/ 6/10 13:40:48 500
n intervals =     100000000 time =    0.105
pi =    3.1415926535897749
difference =   0.0000000000000182
2011/ 6/10 13:40:48 605
n intervals =    1000000000 time =    0.992
pi =    3.1415926535898455
difference =   0.0000000000000524
```

Here is the output from the Cray compiler.

```
 Number of images =  96
2011/ 6/10 13:35: 7 419
n intervals =        100000 time =    0.004
pi =    3.1415926535981265
difference =    0.0000000000083333
2011/ 6/10 13:35: 7 421
n intervals =       1000000 time =    0.000
pi =    3.1415926535898766
difference =   0.0000000000000835
2011/ 6/10 13:35: 7 422
n intervals =      10000000 time =    0.004
pi =    3.1415926535897949
difference =   0.0000000000000018
2011/ 6/10 13:35: 7 424
n intervals =     100000000 time =    0.012
pi =    3.1415926535897913
```

```
 difference =   0.0000000000000018
 2011/ 6/10 13:35: 7 436
 n intervals =    1000000000 time =    0.105
 pi =    3.1415926535897949
 difference =   0.0000000000000018
```

We get the time improvement we have seen with both the MPI and OpenMP solutions.

30.6 Example 4 – Work Sharing

This example looks at one way of splitting work up between images. We use the image number to determine which image does which work. It is a coarray version of the MPI work sharing example.

```
program ch3004
implicit none
integer:: n, i,j
integer:: me, nim, start,end
integer, parameter:: factor=5
integer, dimension(1:factor), codimension[*]:: x
    nim = num_images()
    me  = this_image()
    n   = nim*factor
    x = 0
    start = factor*(me-1) + 1
    end    = factor*me
    j = 1
    do i=start,end
       x(j) = i*factor
       print*,'on image ',me, 'j = ',j,' x(j) = ',x(j)
       j     = j + 1
    end do
    sync all
    if (me == 1) then
      print *,'coarray x on image ',me,' is: ' ,x
      do i=2,nim
        print*, 'coarray x on image ',i,' is: ',x(:)[i]
      end do
    endif
end program ch3004
```

The following statements define the start and end points for the array processing for each image:

```
start = factor*(me-1) + 1
end   = factor*me
```

and partitions the work between the images. Each image will have its own start and end values. The following do loop does the work:

```
do i=start,end
   x(j) = i*factor
   print*,'on image ',me, 'j = ',j,' x(j) = ',x(j)
   j     = j + 1
end do
```

We need the

```
sync all
```

to ensure that each image has completed before further processing, and we then print out the data from each image on image 1.

Here is a subset of the output from the Intel compiler. This example runs on eight images.

```
 on image                2 j =                1  x(j) =
30
 on image                7 j =                1  x(j) =
155
 on image                8 j =                1  x(j) =
180
 on image                8 j =                2  x(j) =
185
 on image                8 j =                3  x(j) =
190
 on image                8 j =                4  x(j) =
195
 on image                8 j =                5  x(j) =
200
 on image                6 j =                1  x(j) =
130
 on image                6 j =                2  x(j) =
135
 on image                6 j =                3  x(j) =
140
...
...
...
...
```

```
              1 j =                5  x(j) =
25
 coarray x on image                1  is:               5
10           15
            20                  25
 on image                   4 j =              1  x(j) =
80
 on image                   4 j =              2  x(j) =
85
 on image                   4 j =              3  x(j) =
90
 on image                   4 j =              4  x(j) =
95
 on image                   4 j =              5  x(j) =
100
 coarray x on image                2  is:              30
35           40
            45                  50
 coarray x on image                3  is:              55
60           65
            70                  75
 coarray x on image                4  is:              80
85           90
            95                 100
 coarray x on image                5  is:             105
110          115
            120                 125
 coarray x on image                6  is:             130
135          140
            145                 150
 coarray x on image                7  is:             155
160          165
            170                 175
 coarray x on image                8  is:             180
185          190
            195                 200
```

Here is a sample of the output from the Cray compiler on the Hector service. This example runs on 96 images.

```
 on image    2 j  =  1  x(j)  =  30
 on image    4 j  =  1  x(j)  =  80
 on image    2 j  =  2  x(j)  =  35
 on image    4 j  =  2  x(j)  =  85
 on image    2 j  =  3  x(j)  =  40
 on image    4 j  =  3  x(j)  =  90
 on image    77 j  =  1  x(j)  =  1905
 on image    74 j  =  1  x(j)  =  1830
 on image    77 j  =  2  x(j)  =  1910
…
…
 on image    64 j  =  2  x(j)  =  1585
 on image    60 j  =  2  x(j)  =  1485
 on image    39 j  =  5  x(j)  =  975
 on image    30 j  =  1  x(j)  =  730
…
…
 on image    31 j  =  1  x(j)  =  755
 on image    42 j  =  1  x(j)  =  1030
 on image    31 j  =  2  x(j)  =  760
…
…
 on image    41 j  =  1  x(j)  =  1005
 on image    27 j  =  3  x(j)  =  665
…
…
 on image    44 j  =  1  x(j)  =  1080
 on image    46 j  =  4  x(j)  =  1145
…
…
 coarray  x  on image   1  is:  5,  10,  15,  20,  25
 coarray  x  on image   2  is:  30,  35,  40,  45,  50
 coarray  x  on image   3  is:  55,  60,  65,  70,  75
 coarray  x  on image   4  is:  80,  85,  90,  95,  100
 coarray  x  on image   5  is:  105,  110,  115,   120,
125
 coarray  x  on image   6  is:  130,  135,  140,   145,
150
 coarray  x  on image   7  is:  155,  160,  165,   170,
175
 coarray  x  on image   8  is:  180,  185,  190,   195,
200
…
…
 coarray  x  on image   88 is:   2180,  2185,  2190,
2195,   2200
```

```
 coarray  x  on image   89 is:   2205,  2210,  2215,
2220,   2225
 coarray  x  on image   90 is:   2230,  2235,  2240,
2245,   2250
 coarray  x  on image   91 is:   2255,  2260,  2265,
2270,   2275
 coarray  x  on image   92 is:   2280,  2285,  2290,
2295,   2300
 coarray  x  on image   93 is:   2305,  2310,  2315,
2320,   2325
 coarray  x  on image   94 is:   2330,  2335,  2340,
2345,   2350
 coarray  x  on image   95 is:   2355,  2360,  2365,
2370,   2375
 coarray  x  on image   96 is:   2380,  2385,  2390,
2395,   2400
```

30.7 Summary

This chapter has looked briefly at some of the simple syntax of coarrays using a small set of examples. We have also seen the timing benefits that coarray programming can offer in the solution of the same problem.

30.8 Problem

1. Compile and run the examples in this chapter with your compiler.

Chapter 31
C Interop

We can't solve problems by using the same kind of thinking we used when we created them.

Einstein

Aim

This chapter looks briefly at C interoperability.

31.1 Introduction

C is a widely used programming languages and there is a considerable amount of software written in C. Fortran 2003 introduced a standardised mechanism for interoperating with C.

In this chapter we provide a brief coverage of some of the technical details required for interoperability and then have a look at a couple of examples.

31.2 ISO_C_BINDING Module

There is an intrinsic module called ISO_C_BINDING that contains named constants, derived types and module procedures to support interoperability.

31.3 Named Constants and Derived Types in the Module

The entities listed in the second column of Table 31.1, are named constants of type default integer.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_31, © Springer-Verlag London Limited 2012

Table 31.1

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
INTEGER	C_INT	Int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char
		unsigned char
	C_SIZE_T	size t
	C_INT8_T	int8 t
	C_INT16_T	int16 t
	C_INT32_T	int32 t
	C_INT64_T	int64 t
	C_INT_LEAST8_T	int least8 t
	C_INT_LEAST16_T	int least 16 t
	C_INT_LEAST32_T	int least32 t
	C_INT_LEAST64_T	int least64 t
	C_INT_FAST8_T	int fast8 t
	C_INT_FAST16_T	int fast16 t
	C_INT_FAST32_T	int fast32 t
	C_INT_FAST64_T	int fast64 t
	C_INTMAX_T	intmax t
	C_INTPTR_T	intptr t
REAL	C_FLOAT	Float
	C_DOUBLE	Double
	C_LONG_DOUBLE	long double
COMPLEX	COMPLEX_C_DOUBLE _COMPLEX	Double Complex
	C_LONG_DOUBLE_COMPLEX	long double Complex
LOGICAL	C_BOOL	Bool
CHARACTER	C_CHAR	char

The above mentioned C types are defined in the C International Standard, clauses 6.2.5, 7.17, and 7.18.1

31.4 Character Interoperability

The following maps between Fortran and C character types. The semantics of these values are explained in 5.2.1 and 5.2.2 of the C International Standard.

Names of C characters with special semantics			
Name	C definition	C_CHAR =−1	C_CHAR/=−1
C_NULL_CHAR	Null character	CHAR(0)	'\0'
C_ALERT	Alert	ACHAR(7)	'\a'
C_BACKSPACE	Backspace	ACHAR(8)	'\b'
C_FORM_FEED	Form feed	ACHAR(12)	'\f'
C_NEW_LINE	New line	ACHAR(10)	'\n'
C_CARRIAGE_RETURN	Carriage return	ACHAR(13)	'\r'
C_HORIZONTAL_TAB	Horizontal tab	ACHAR(9)	'\t'
C_VERTICAL_TAB	Vertical tab	ACHAR(11)	'\v'

31.5 Procedures in the Module

There are several procedures in this module. In the descriptions below, procedure names are generic and not specific.

A C procedure argument is often defined in terms of a C address. The C_LOC and C_FUNLOC functions are provided so that Fortran applications can determine the appropriate value to use with C facilities.

The C_ASSOCIATED function is provided so that Fortran programs can compare C addresses.

The C_F_POINTER and C_F_PROCPOINTER subroutines provide a means of associating a Fortran pointer with the target of a C pointer.

More information can be found in Chap. 15 of the standard.

31.6 Interoperability of Intrinsic Types

Table 31.1 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic type with particular type parameter values is interoperable with a C type if the type and kind type parameter value are listed in the table on the same row as that C type; if the type is character, interoperability also requires that the length type parameter be omitted or be specified by an initialization expression whose value is one. A combination of Fortran type and type parameters that is interoperable with a C type listed in the table is also interoperable with any unqualified C type that is compatible with the listed C type.

The second column of the table refers to the named constants made accessible by the ISO_C_BINDING intrinsic module.

A combination of intrinsic type and type parameters is interoperable if it is interoperable with a C type.

31.7 Other Aspects of Interoperability

There are considerable restrictions on other aspects of interoperability. The following provides some brief details of other areas:

- Interoperability with C pointer types
  - C_PTR and C_FUNPTR shall be derived types with private components. C_PTR is interoperable with any C object pointer type. C_FUNPTR is interoperable with any C function pointer type.
- Interoperability of scalar variables
  - A scalar Fortran variable is interoperable if its type and type parameters are interoperable and it has neither the pointer nor the allocatable attribute.
  - An interoperable scalar Fortran variable is interoperable with a scalar C entity if their types and type parameters are interoperable.

- Interoperability of array variables
  - An array Fortran variable is interoperable if its type and type parameters are interoperable and it is of explicit shape or assumed size.
- Interoperability of procedures and procedure interfaces
  - A Fortran procedure is interoperable if it has the BIND attribute, that is, if its interface is specified with a proc-language-binding-spec.
- Interoperation with C global variables
  - AC variable with external linkage may interoperate with a common block or with a variable declared in the scope of a module. The common block or variable shall be specified to have the BIND attribute.
- Binding labels for common blocks and variables
  - The binding label of a variable or common block is a value of type default character that specifies the name by which the variable or common block is known to the companion processor.
- Interoperation with C functions
  - A procedure that is interoperable may be defined either by means other than Fortran or by means of a Fortran subprogram, but not both.

Another useful source can be found in the December 2009 edition of Fortran Forum. Details are given at the end of the chapter.

31.8 C_LOC Examples

We include a small number of examples using the C_LOC function. Here is some of the technical information on C_LOC from the standard.

C_LOC (X)

- Description.
  Returns the C address of the argument.
- Class.
  Inquiry function.
- Argument.
  X shall either
  1. have interoperable type and type parameters and be
     (a) a variable that has the TARGET attribute and is interoperable,
     (b) an allocated allocatable variable that has the TARGET attribute and is not an array of zero size, or
     (c) an associated scalar pointer, or

2. be a nonpolymorphic scalar, have no length type parameters, and be

   (a) a nonallocatable, nonpointer variable that has the TARGET attribute,
   (b) an allocated allocatable variable that has the TARGET attribute, or
   (c) an associated pointer.

- Result Characteristics.
  Scalar of type C_PTR.
- Result Value.

  The result value will be described using the result name CPTR.

  1. If X is a scalar data entity, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of X and the pointer assignment CPTR%PX => X were executed.
  2. If X is an array data entity, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of X and the pointer assignment of CPTR%PX to the first element of X were executed.

- If X is a data entity that is interoperable or has interoperable type and type parameters, the result is the value that the C processor returns as the result of applying the unary "&" operator (as defined in the C International Standard, 6.5.3.2) to the target of CPTR
- The result is a value that can be used as an actual CPTR argument in a call to C_F_POINTER where FPTR has attributes that would allow the pointer assignment FPTR => X. Such a call to C_F_POINTER shall have the effect of the pointer assignment FPTR => X.
- NOTE 15.6 – Where the actual argument is of noninteroperable type or type parameters, the result of C_LOC provides an opaque "handle" for it. In an actual implementation, this handle may be the C address of the argument; however, portable C functions should treat it as a void (generic) C pointer that cannot be dereferenced (6.5.3.2 in the C International Standard).

The key issues are that we must take care with the argument to the function, the return value is of type C_PTR, and that this is an opaque type. Let us now look at some examples using this function.

31.9 Example 1

The arguments x1 and x2 to c_loc are variables with the target attribute. The arguments p_x1 and p_x2 are both pointers.

The return values from the c_loc function must be of type c_ptr. In the first call to c_loc these pointers are not associated.

We can't print variables of type c_ptr so we use the transfer intrinsic to convert to an integer value that we can print.

```
program ch3101
use iso_c_binding
implicit none

integer , target   :: x1 = 20
integer , target   :: x2 = 30
integer , pointer  :: p_x1
integer , pointer  :: p_x2

type (c_ptr)        :: c_ptr1,c_ptr2,c_ptr3,c_ptr4

integer :: i1,i2,i3,i4
  c_ptr1=c_loc(x1)
  c_ptr2=c_loc(x2)
  c_ptr3=c_loc(p_x1)
  c_ptr4=c_loc(p_x2)

  i1=transfer(c_ptr1,i1)
  i2=transfer(c_ptr2,i2)
  i3=transfer(c_ptr3,i3)
  i4=transfer(c_ptr4,i4)

  print *,' i1 ',i1
  print *,' i2 ',i2
  print *,' i3 ',i3
  print *,' i4 ',i4

  p_x1 => x1
  p_x2 => x2

  c_ptr3=c_loc(p_x1)
  c_ptr4=c_loc(p_x2)

  i1=transfer(c_ptr1,i1)
  i2=transfer(c_ptr2,i2)
  i3=transfer(c_ptr3,i3)
  i4=transfer(c_ptr4,i4)

  print *, ' x1 ',x1
  print *, ' x2 ',x2
  print *, ' p_x1 ',p_x1
  print *, ' p_x2 ',p_x2
  print *,' i1 ',i1
  print *,' i2 ',i2
  print *,' i3 ',i3
  print *,' i4 ',i4

end program ch3101
```

Here is the output from three compilers.

31.9.1 Gfortran Output

```
i1         4374528
i2         4374532
i3               0
i4               0
x1              20
x2              30
p_x1              20
p_x2              30
i1         4374528
i2         4374532
i3         4374528
i4         4374532
```

The program prints out 0 for i3 and i4 initially. At this point the pointers c_ptr3 and c_ptr4 are not associated, i.e. they do not point to anything.

31.9.2 Intel Output

```
i1      1065910272
i2      1065910276
i3               0
i4               0
x1              20
x2              30
p_x1              20
p_x2              30
i1      1065910272
i2      1065910276
i3      1065910272
i4      1065910276
```

Intel does something similar to gfortran and the program prints out 0 for i3 and i4 initially. At this point the pointers c_ptr3 and c_ptr4 are not associated, i.e. they do not point to anything.

31.9.3 Nag Output

```
 i1   4206592
 i2   4206596
 i3   -1
 i4   -1
 x1   20
 x2   30
 p_x1   20
 p_x2   30
 i1   4206592
 i2   4206596
 i3   4206592
 i4   4206596
```

The Nag program prints out −1 for i3 and i4 initially. At this point the pointers c_ptr3 and c_ptr4 are not associated, i.e. they do not point to anything.

The value zero is often used to signify a special memory value in computing and this is chosen by the gfortran and Intel compilers. The Nag compiler chooses −1, again a special value.

31.10 Example 2

In this example we use the `null ()` intrinsic to provide initial values for the two pointer variables p_x1 and p_x2.

```
program ch3102
use iso_c_binding
implicit none

integer , target   :: x1 = 20
integer , target   :: x2 = 30
integer , pointer  :: p_x1=>null()
integer , pointer  :: p_x2=>null()

type (c_ptr)        :: c_ptr1,c_ptr2,c_ptr3,c_ptr4

integer :: i1,i2,i3,i4

  c_ptr1=c_loc(x1)
  c_ptr2=c_loc(x2)
  c_ptr3=c_loc(p_x1)
  c_ptr4=c_loc(p_x2)
```

```
  i1=transfer(c_ptr1,i1)
  i2=transfer(c_ptr2,i2)
  i3=transfer(c_ptr3,i3)
  i4=transfer(c_ptr4,i4)

  print *, ' i1 ',i1
  print *, ' i2 ',i2
  print *, ' i3 ',i3
  print *, ' i4 ',i4

  p_x1 => x1
  p_x2 => x2

  c_ptr3=c_loc(p_x1)
  c_ptr4=c_loc(p_x2)

  i1=transfer(c_ptr1,i1)
  i2=transfer(c_ptr2,i2)
  i3=transfer(c_ptr3,i3)
  i4=transfer(c_ptr4,i4)

  print *, ' x1 ',x1
  print *, ' x2 ',x2
  print *, ' P_x1 ',p_x1
  print *, ' P_x2 ',p_x2
  print *,' il ',il
  print *,' i2 ',i2
  print *,' i3 ',i3
  print *,' i4 ',i4

end program ch3102
```

The output for the gfortran and Intel compilers is as before. The Nag output is given below.

```
  i1   4206592
  i2   4206596
  i3   0
  i4   0
  x1   20
  x2   30
  p_x1   20
  p_x2   30
  i1   4206592
  i2   4206596
  i3   4206592
  i4   4206596
```

The Nag compiler therefore distinguishes between pointers that are uninitialised (−1) and initialised (0) to the null value, i.e. not associated. This kind of bug is quite hard to find!

31.11 Bibliography

Einarsson, B., Hanson, R.J., Hopkins, T.: Standardized mixed language programming for Fortran and C. Fortran Forum, **28**(3), (December 2009)

31.12 Problem

1. Compile and run the example programs in this chapter with your compiler and examine the output.

Chapter 32
ISO TR 15580 IEEE Arithmetic

Any effectively generated theory capable of expressing elementary arithmetic cannot be both consistent and complete. In particular, for any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory.

Godel, First incompleteness theorem

Aims

The aims of this chapter are to look in more depth at arithmetic and in particular at the support that Fortran provides for the IEEE 754 standard. There is a coverage of:

- Hardware support for arithmetic.
- Integer formats.
- Floating point formats: single and double.
- Special values: denormal, infinity and not a number – NAN.
- Exceptions and flags: divide by zero, inexact, invalid, overflow, under flow.

32.1 Introduction

The literature contains details of the IEEE 754 standard and the bibliography contains details of a number of printed and on-line sources.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_32, © Springer-Verlag London Limited 2012

32.2 History

When we use programming languages to do arithmetic two major concerns are the ability to develop reliable and portable numerical software. Arithmetic is done in hardware and there are a number of things to consider:

- The range of hardware available both now and in the past.
- The evolution of hardware.

There has been a very considerable change in arithmetic units since the first computers. The following is a list of hardware and computing systems that the authors have used or have heard of. It is not exhaustive or definitive, but rather reflects the authors' age and experience:

- CDC
- Cray
- IBM
- ICL
- Fujitsu
- DEC
- Compaq
- Gateway
- Sun
- Silicon Graphics
- Hewlett Packard
- Data General
- Honeywell
- Elliot
- Mostek
- National Semiconductors
- Intel
- Zilog
- Motorola
- Signetics
- Amdahl
- Texas Instruments
- Cyrix
- AMD

Some of the operating systems include:

- NOS
- NOS/BE
- Kronos
- UNIX
- VMS
- Dos
- Windows 3.x

- Windows 95
- Windows 98
- Windows NT
- Windows 2000
- Windows XP
- Windows Vista
- MVS
- VM
- CP/M
- Macintosh
- OS/2
- Linux, a multitude!

Again the list is not exhaustive or definitive. The intention is simply to provide some idea of the wide range of hardware, computer manufacturers and operating systems that have been around in the past 50 years.

To cope with the anarchy in this area Doctor Robert Stewart (acting on behalf of the IEEE) convened a meeting which led to the birth of IEEE 754.

The first draft, which was prepared by William Kahan, Jerome Coonen and Harold Stone, was called the KCS draft and eventually adopted as IEEE 754. A fascinating account of the development of this standard can be found in An Interview with the Old Man of Floating Point, and the bibliography provides a web address for this interview. Kahan went on to get the ACM Turing Award in 1989 for his work in this area.

This has become a de facto standard amongst arithmetic units in modern hardware. Note that it is not possible to describe precisely the answers a program will give, and the authors of the standard knew this. This goal is virtually impossible to achieve when one considers floating point arithmetic. Reasons for this include:

- The conversions of numbers between decimal and binary formats.
- The use of elementary library functions.
- Results of calculations may be in hardware inaccessible to the programmer.
- Intermediate results in subexpressions or arguments to procedures.

The bibliography contains details of a paper that addresses this issue in much greater depth – Differences Among IEEE 754 Implementations.

Fortran is one of a small number of languages that provides access to IEEE arithmetic, and it achieves this via TR1880 which is an integral part of Fortran 2003. The C standard (C9X) addresses this issue and Java offers limited IEEE arithmetic support. More information can be found in the references at the end of the chapter.

32.3 IEEE 754 Specifications

The standard specifies a number of things including:

- Single precision floating point format.
- Double precision floating point format.

- Two classes of extended floating point formats.
- Accuracy requirements on the following floating point operations:
  - Add.
  - Subtract.
  - Multiply.
  - Divide.
  - Square root.
  - Remainder.
  - Round numbers in floating point format to integer values.
  - Convert between different floating point formats.
  - Convert between floating point and integer format.
  - Compare.
- Base conversion, i.e., when converting between decimal and binary floating point formats and vice versa.
- Exception handling for:
  - Divide by zero.
  - Overflow.
  - Underflow.
  - Invalid operation.
  - Inexact.
- Rounding directions.
- Rounding precisions.

We will look briefly at each of these requirements.

32.3.1 Single Precision Floating Point Format

This is a 32-bit quantity made up of a sign bit, 8-bit biased exponent and 23-bit mantissa. The standard also specifies that certain of the bit patterns are set aside and do not represent normal numbers. This means that valid numbers are in the range 3.40282347E + 38 to 1.17549435E-38 and the precision is between 6 and 9 digits depending on the numbers.

The special bit patterns provide the following:

- +0
- –0
- subnormal numbers in the range 1.17549421E-38 to 1.40129846E-45
- + infinity
- – infinity
- quiet NaN (Not a Number)
- signalling NaN

One of the first systems that the authors worked with that had special bit patterns set aside was the CDC 6000 range of computers that had negative indefinite and infinity. Thus the ideas are not new, as this was in the late 1970s.

The support of positive and negative zero means that certain problems can be handled correctly including:

- The evaluation of the log function which has a discontinuity at zero.
- The equation $\sqrt{1/z} = 1/z$ can be solved when $z = -1$.

See also the Kahan paper Branch Cuts for complex Elementary Functions, or Much Ado About Nothing's Sign Bit for more details.

Subnormals, which permit gradual underflow, fill the gap between 0 and the smallest normal number.

Simply stated underflow occurs when the result of an arithmetic operation is so small that it is subject to a larger than normal rounding error when stored. The existence of subnormals means that greater precision is available with these small numbers than with normal numbers. The key features of gradual underflow are:

- When underflow does occur there should never be a loss of accuracy any greater than that from ordinary roundoff.
- The operations of addition, subtraction, comparison and remainder are always exact.
- Algorithms written to take advantage of subnormal numbers have smaller error bounds than other systems.
- if x and y are within a factor of 2 then x-y is error free, which is used in a number of algorithms that increase the precision at critical regions.

The combination of positive and negative zero and subnormal numbers means that when x and y are small and x-y has been flushed to zero the evaluation of

- $1/(x-y)$

can be flagged and located.

Certain arithmetic operations cause problems including:

- $0 * \infty$
- 0/0
- $\sqrt{x}$ when $x < 0$

and the support for NaN handles these cases.

The support for positive and negative infinity allows the handling of

- x/0 when x is nonzero and of either sign

and the outcome of this means that we write our programs to take the appropriate action. In some cases this would mean recalculating using another approach.

For more information see the references in the bibliography.

32.3.2 Double Precision Floating Point Format

This is a 64-bit quantity made up of a sign bit, 11-bit biased exponent and 52-bit mantissa. As with single precision the standard specifies that certain of the bit patterns are set aside and do not represent normal numbers. This means we have valid numbers in the range 1.7976931348623157E308 to 2.2250738585072014E-308 and precision between 15 and 17 digits depending on the numbers.

As with single precision there are bit patterns set aside for the same special conditions.

Note that this does not mean that the hardware has to handle the manipulation of this 64-bit quantity in an identical fashion. The Sparc and Intel family handle the above as two 32-bit quantities but the order of the two component parts is reversed – so-called big endian and little endian.

32.3.3 Two Classes of Extended Floating Point Formats

These formats are not mandatory. A number of variants of double extended exist including:

- Sun – four 32-bit words, one sign bit, 15-bit biased exponent and 112-bit mantissa, numbers in the range 3.362E-4932 to 1.189E4932, 33–36 digits of significance.
- Intel – 10 bytes – one sign bit, 15-bit biased exponent, 63-bit mantissa, numbers in the range 3.362E-4932 to 1.189E4932, 18–21 digits of significance.
- PowerPC – as Sun.

32.3.4 Accuracy Requirements

Remainder and compare must be exact. The rest should return the exact result if possible; if not, there are well-defined rounding rules to apply.

32.3.5 Base Conversion – Converting Between Decimal and Binary Floating Point Formats and Vice Versa

These results should be exact if possible; if not the results must differ by tolerances that depend on the rounding mode.

32.3.6 Exception Handling

It must be possible to signal to the user the occurrence of the following conditions or exceptions:

- Divide by zero.
- Overflow.
- Underflow.
- Invalid operation.
- Inexact.

The ability to detect the above is a big step forward in our ability to write robust and portable code. These operations do occur in calculations and it is essential to have user programmer control over what action to take.

32.3.7 Rounding Directions

Four rounding directions are available:

- Nearest – the default.
- Down.
- Up.
- Chop.

Access to directed rounding can be used to implement interval arithmetic, for example.

32.3.8 Rounding Precisions

The only mandatory part here is that machines that perform computations in extended mode let the programmer control the precision via a control word. This means that if software is being developed on machines that support extended modes those machines can be switched to a mode that would enable the software to run on a system that didn't support extended modes. This area looks like a can of worms. Look at the Kahan paper for more information – Lecture Notes on the Status of IEEE 754.

32.4 Resumé

The above has provided a quick tour of IEEE 754. We'll now look at what Fortran has to offer to support it.

32.5 ISO TR 15580

Fortran provides access to the facilities via the use statement. The current standard does not have the concept of an intrinsic module. TR 15580 introduces this concept. Three modules are provided:

- ieee_features
- ieee_exceptions
- ieee_arithmetic

The first thing to consider is the degree of conformance to the IEEE standard. It is possible that not all of the features are supported. Thus the first thing to do is to run one or more test programs to determine the degree of support for a particular system.

32.5.1 IEEE_FEATURES Module

This module defines a derived type, IEEE_FEATURES_type, and up to 11 constants of that type representing IEEE features:

- IEEE_DATATYPE – whether any IEEE data types are available.
- IEEE_DENORMAL – whether IEEE denormal values are available.
- IEEE_DIVIDE – whether division has the accuracy required by IEEE.
- IEEE_HALTING – whether control of halting is supported.
- IEEE_INEXACT_FLAG – whether the inexact exception is supported.
- IEEE_INF – whether IEEE positive and negative infinities are available.
- IEEE_INVALID_FLAG – whether the invalid exception is supported.
- IEEE_NAN – whether IEEE NaNs are available.
- IEEE_ROUNDING – whether all IEEE rounding modes are available.
- IEEE_SQRT – whether SQRT conforms to the IEEE standard.
- IEEE_UNDERFLOW_FLAG – whether underflow is supported.

32.5.2 IEEE_EXCEPTIONS Module

This module provides data types, constants and generic procedures for IEEE exceptions:

type IEEE_STATUS_TYPE

Variables of this type can hold a floating point status value.

subroutine IEEE_GET_STATUS(STATUS_VALUE)

type(IEEE_STATUS_TYPE),intent(out) :: STATUS_VALUE

Stores the current floating point status into the STATUS_VALUE argument.

subroutine IEEE_SET_STATUS(STATUS_VALUE)

type(IEEE_STATUS_TYPE),intent(in) :: STATUS_VALUE

Sets the current floating point status from the STATUS_VALUE argument.

type IEEE_FLAG_TYPE

Values of this type specify individual IEEE exception flags; constants for these are available as follows:

type(IEEE_FLAG_TYPE),parameter :: IEEE_DIVIDE_BY_ZERO

type(IEEE_FLAG_TYPE),parameter :: IEEE_INEXACT

type(IEEE_FLAG_TYPE),parameter :: IEEE_INVALID

type(IEEE_FLAG_TYPE),parameter :: IEEE_OVERFLOW

type(IEEE_FLAG_TYPE),parameter :: IEEE_UNDERFLOW

In addition, two array constants are available for indicating common combinations of flags:

type(IEEE_FLAG_TYPE),parameter :: &

IEEE_USUAL(3) = (/&

IEEE_DIVIDE_BY_ZERO,&

IEEE_INVALID, &

IEEE_OVERFLOW /), &

IEEE_ALL(5) = (/&

IEEE_DIVIDE_BY_ZERO,&

IEEE_INEXACT, &

IEEE_INVALID,&

IEEE_OVERFLOW, &

IEEE_UNDERFLOW /)

LOGICAL function IEEE_SUPPORT_FLAG(FLAG,X)

type(IEEE_FLAG_TYPE),intent(in) :: FLAG

real(kind),intent(in),optional :: X

Returns TRUE if detection of the specified IEEE exception is supported for the real kind of X (if X is present), or for all real kinds (if X is absent).

LOGICAL function IEEE_SUPPORT_HALTING(FLAG)

type(IEEE_FLAG_TYPE),intent(in) :: FLAG

Returns TRUE if IEEE_SET_HALTING_MODE can be used to change whether the processor terminates the program on receiving the specified exception.

elemental subroutine &
IEEE_GET_FLAG(FLAG,FLAG_VALUE)

type(IEEE_FLAG_TYPE),intent(in) :: FLAG

LOGICAL,intent(out) :: FLAG_VALUE

Sets (each element of) FLAG_VALUE to TRUE if the corresponding exception specified by FLAG is signalling, and to FALSE otherwise.

elemental subroutine &
IEEE_GET_HALTING_MODE(FLAG,HALTING)

type(IEEE_FLAG_TYPE),intent(in) :: FLAG

LOGICAL,intent(out) :: HALTING

Sets (each element of) HALTING to TRUE if the corresponding exception specified by FLAG is signalling, and to FALSE otherwise.

elemental subroutine IEEE_SET_FLAG(FLAG,FLAG_VALUE)

type(IEEE_FLAG_TYPE),intent(out) :: FLAG

LOGICAL,intent(in) :: FLAG_VALUE

Sets the exception flag specified by (each element of) FLAG to signalling or quiet according to the corresponding element of FLAG_VALUE.

elemental subroutine &
IEEE_SET_HALTING_MODE(FLAG,HALTING)

type(IEEE_FLAG_TYPE),intent(out) :: FLAG

LOGICAL,intent(in) :: HALTING

Sets the halting mode for each exception specified by FLAG to the value of the corresponding element of HALTING (TRUE = halt).

32.5.3 IEEE_ARITHMETIC Module

These are given below

32.5.3.1 IEEE Data Type Selection

Integer Function SELECTED_real_KIND(P,R)
integer(kind1),optional :: P

integer(kind2),optional :: R

The same as the SELECTED_real_KIND intrinsic, but only returns information about the IEEE kinds of reals.

32.5.3.2 General Support Enquiry Functions

LOGICAL function IEEE_SUPPORT_DATATYPE(X)

real(kind),optional :: X

Whether IEEE arithmetic is supported for the same kind of real as X (or for all real kinds if X is absent).

LOGICAL function IEEE_SUPPORT_DENORMAL(X)

real(kind),optional :: X

Whether IEEE denormal values are supported for the same kind of real as X (or for all real kinds if X is absent).

LOGICAL function IEEE_SUPPORT_DIVIDE(X)

real(kind),optional :: X

Whether division is carried out to the accuracy specified by the IEEE standard for the same kind of real as X (or for all real kinds if X is absent).

LOGICAL function IEEE_SUPPORT_INF(X)

real(kind),optional :: X

Whether IEEE infinite values are supported for the same kind of real as X (or for all real kinds if X is absent).

LOGICAL function IEEE_SUPPORT_NAN(X)

real(kind),optional :: X

Whether IEEE NaN (Not-a-Number) values are supported for the same kind of real as X (or for all real kinds if X is absent).

LOGICAL function IEEE_SUPPORT_SQRT(X)

real(kind),optional :: X

Whether SQRT conforms to the IEEE standard for the same kind of real as X (or for all real kinds if X is absent).

LOGICAL function IEEE_SUPPORT_STANDARD(X)

real(kind),optional :: X

Whether all the IEEE facilities specified by the TR are supported for the same kind of real as X (or for all real kinds if X is absent).

32.5.3.3 Rounding Modes

type IEEE_ROUND_type

Values of this type specify the IEEE rounding mode.

type (IEEE_ROUND_TYPE) , parameter :: IEEE_DOWN

type (IEEE_ROUND_TYPE) , parameter :: IEEE_NEAREST

type (IEEE_ROUND_TYPE) , parameter :: IEEE_TO_ZERO

type (IEEE_ROUND_TYPE) , parameter :: IEEE_UP

LOGICAL function IEEE_SUPPORT_ROUNDING(ROUND_VALUE,X)
type(IEEE_ROUND_type),intent(in) :: ROUND_VALUE

real(kind),optional :: X

Whether the specified IEEE rounding mode is supported for the same kind of real as X (or for all real kinds if X is absent).

subroutine IEEE_GET_ROUNDING_MODE(ROUND_VALUE)

type(IEEE_ROUND_TYPE),intent(out) :: ROUND_VALUE

Sets the ROUND_VALUE argument to the current IEEE rounding mode.

subroutine IEEE_SET_ROUNDING_MODE(ROUND_VALUE)

type (IEEE_ROUND_TYPE) , intent(in) :: ROUND_VALUE

Sets the current IEEE rounding mode to that specified by ROUND_VALUE.

32.5.3.4 Number Classification

Type IEEE_CLASS_TYPE

Values of this type indicate the IEEE class of a number.

type (IEEE_CLASS_TYPE) , &

parameter :: IEEE_NEGATIVE_DENORMAL

type (IEEE_CLASS_TYPE) , parameter:: IEEE_NEGATIVE_INF

type (IEEE_CLASS_TYPE) , parameter:: IEEE_NEGATIVE_NORMAL

type (IEEE_CLASS_TYPE) , parameter:: IEEE_NEGATIVE_ZERO

type (IEEE_CLASS_TYPE) , parameter:: IEEE_POSITIVE_DENORMAL

type (IEEE_CLASS_TYPE) , parameter:: IEEE_POSITIVE_INF

type (IEEE_CLASS_TYPE) , parameter:: IEEE_POSITIVE_NORMAL

type (IEEE_CLASS_TYPE) , parameter:: IEEE_POSITIVE_ZERO

type (IEEE_CLASS_TYPE) , parameter:: IEEE_QUIET_NAN

type (IEEE_CLASS_TYPE) , parameter:: IEEE_signalling_NAN

elemental type(IEEE_CLASS_TYPE) function IEEE_class(X)

real(kind),intent(in) :: X

Returns the appropriate value of IEEE_CLASS_TYPE for the number X, which may be of any IEEE kind.

In addition to ISO/IEC TR 15580:1998(E), the module IEEE_ARITHMETIC defines the "==" and "/=" operators for the IEEE_CLASS_TYPE. These may be used to test the return value of the IEEE_class function, e.g.,

use,intrinsic :: IEEE_ARITHMETIC, only: IEEE_class, &

IEEE_QUIET_NAN, operator(==)

. . .

if (IEEE_class(X)== IEEE_QUIET_NAN) then

. . .

elemental real(kind) function IEEE_VALUE(X,class)

real(kind),intent(in) :: X

type(IEEE_CLASS_TYPE),intent(in) :: class

Returns a sample value of the specified class for the same kind of real as X, which may be of any IEEE kind.

elemental LOGICAL function IEEE_IS_FINITE(X)

real(kind),intent(in) :: X

Returns TRUE if X is not infinite or NaN.

elemental LOGICAL function IEEE_IS_NAN(X)

real(kind),intent(in) :: X

Returns TRUE if X is either a signalling or quiet NaN.

elemental LOGICAL function IEEE_IS_NEGATIVE(X)

real(kind),intent(in) :: X

Returns TRUE if X is negative, including negative zero.

elemental LOGICAL function IEEE_IS_NORMAL(X)

real(kind),intent(in) :: X

Returns TRUE if X is not an infinity, NaN, or denormal.

elemental LOGICAL function IEEE_UNORDERED(X,Y)

real(kind),intent(in) :: X,Y

Returns TRUE if X is a NaN or if Y is a NaN.

32.5.3.5 Arithmetic Operations

elemental real(kind) function IEEE_COPY_SIGN(X,Y)

real (kind) , intent(in) :: X,Y

Returns X with the sign of Y, even for NaNs and infinities.

elemental real (kind) function IEEE_LOGB(X)

real (kind) , intent(in) :: X

Returns the unbiased exponent as a real value:

if X is zero, IEEE_DIVIDE_BY_ZERO signals and the result is –infinity if IEEE infinities are supported for that kind, and –HUGE(X) if not.

if X is infinite, the result is + infinity.

if X is a NaN, the result is a quiet NaN (the same one if X is a quiet NaN); otherwise the result is EXPONENT(X)-1.

elemental real (kind) function IEEE_NEXT_AFTER(X,Y)

real (kind) , intent(in) :: X,Y

The same as NEAREST(X,1.0_kind) for Y > X and NEAREST(X,-1.0_kind) for Y < X; if Y==X, the result is X, if either X or Y are NaNs the result is one of these NaNs.

elemental real (kind) function IEEE_REM(X,Y)

real (kind) , intent(in) :: X,Y

X-Y*N exactly, where N is the integer nearest to the exact value X/Y. if the result is zero, it has the same sign as X. This function is not affected by the rounding mode.

elemental real (kind) function IEEE_RINT(X)

real (kind) , intent(in) :: X

Round to an integer according to the current rounding mode.

elemental real (kind) function IEEE_SCALB(X,I)

real (kind1) , intent(in) :: X

integer (kind2) , intent(in) :: I

The same as SCALE(X,I).

32.6 Summary

Support for the above is relatively limited at the time of writing this book. There is always a time lag between the formal publication of a standard and the implementation in production compilers. As compiler support improves examples will be added to our web site. Our home page is:

- http://www.fortranplus.co.uk/

32.7 Bibliography

Hauser, J.R.: Handling floating point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (1996)

- The paper looks at a number of techniques for handling floating point exceptions in numeric code. One of the conclusions is for better structured support for floating point exception handling in new programming languages, or of course better standards for existing languages.

IEEE, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-2008, Institute of Electrical and Electronic Engineers Inc.

The formal definition of IEEE 754. This is available for purchase at http://www.techstreet.com/standards/ieee/754_2008?product_id=1745167 as both a pdf and printed version.

This standard specifies formats and methods for floating-point arithmetic in computer systems: standard and extended functions with single, double, extended, and extendable precision, and recommends formats for data interchange. Exception conditions are defined and standard handling of these conditions is specified. Keywords: 754-2008,arithmetic,binary,computer,decimal, exponent, floating-point, format,interchange,NaN,number,rounding,significand,subnormal. Product Code(s): STDPD95802,STD95802

Knuth, D.: Seminumerical Algorithms. Addison-Wesley, Reading (1969)

- There is a coverage of floating point arithmetic, multiple precision arithmetic, radix conversion and rational arithmetic.

Sun: Numerical Computation Guide. SunPro, Mountain View (1995)

- Very good coverage of the numeric formats for IEEE Standard 754 for Binary Floating-Point Arithmetic. All SunPro compiler products support the features of the IEEE 754 standard.

32.7.1 Web-Based Sources

http://validgh.com/goldberg/addendum.html

- Differences Among IEEE 754 Implementations. The material in this paper will eventually be included in the Sun Numerical Computation Guide as an addendum to Appendix C, David Goldberg's What Every Computer Scientist Should Know about Floating Point Arithmetic.

http://docs.sun.com/

- Follow the links to the Floating Point and common Tools AnswerBook. The Numerical Computation Guide can be browsed on-line or down loaded as a pdf file. The last time we checked it was about 260 pages. Good source of information if you have Sun equipment.

http://www.validgh.com/

- This web site contains technical and business information relating to the validgh professional consulting practice of David G. Hough. Contains links to the Goldberg paper and the above addendum by Doug Priest.

http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html

- Brief coverage of IEEE arithmetic with pointers to further sources. There is also a coverage of the storage layout and ranges of floating point numbers. Computer Science 341 is an introduction to the design of a computer's hardware, particularly the CPU and memory systems.

http://www.nag.co.uk/nagware/NP/TR.html

- NAG provide coverage of TR 15580 and TR 15581. The first is the support Fortran has for IEEE arithmetic.

http://www.cs.berkeley.edu/~wkahan/

- Willam Kahan home page.

http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

- An Interview with the Old Man of Floating Point. Reminiscences elicited from William Kahan by Charles Severance, which appeared in an issue of IEEE Computer – March 1998.

http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

- Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic. Well worth a read.

http://www.stewart.cs.sdsu.edu/cs575/labs/l3floatpt.html

- CS 575 Supercomputing – Lab 3: Floating Point Arithmetic. CS 575 is an interdisciplinary course to introduce students in the sciences and engineering to advanced computing techniques using the supercomputers at the San Diego Supercomputer Center (SDSC).

http://www.mathcom.com/nafaq/index.html

- FAQ: Numerical Analysis and Associated Fields Resource Guide. A summary of Internet resources for a number of fields related to numerical analysis.

http://www.math.psu.edu/dna/disasters/ariadne.html

- The Explosion of the Ariane 5: A 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. The number was larger than 32,768, the largest integer storeable in a 16-bit signed integer, and thus the conversion failed.

32.7.2 Hardware Sources

Amd

Visit

http://developer.amd.com/documentation/guides/pages/default.aspx

for details of the AMD manuals. The following five manuals are available for download as pdf s from the above site.

- AMD64 Architecture Programmer's Manual Volume 1: Application Programming
- AMD64 Architecture Programmer's Manual Volume 2: System Programming
- AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions
- AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256 bit media instructions
- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions

Intel

Visit

http://www.intel.com/products/processor/manuals/index.htm

for a list of manuals. The following three manuals are available for download as pdf s from the above site.

- Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture
- Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 2A and 2B: Instruction Set Reference, A-Z.
- Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2

Osbourne, A., Kane, G.: 4-bit and 8-bit Microprocessor Handbook. Osbourne/McGraw-Hill, Berkeley (1981)

- Good source of information on 4-bit and 8-bit microprocessors.

Osbourne, A., Kane, G.: 16-Bit Microprocessor Handbook. Osbourne/McGraw-Hill, Berkely (1981)

- Ditto 16-bit microprocessors.

Bhandarkar, D.P.: Alpha Implementations and Architecture: Complete Reference and Guide. Digital Press, Boston (1996)

- Looks at some of the trade-offs and design philosophy behind the alpha chip. The author worked with VAX, Micro VAX and VAX vectors as well as the Prism. Also looks at the GEM compiler technology that DEC/Compaq use.

http://www.digital.com/alphaserver/workstations/

- Home page for the Compaq/DEC Alpha systems.

http://www.sgi.com/

- Silicon Graphics home page.

http://www.sun.com/

- Sun home page.

http://www.ibm.com/

- IBM home page.

32.7.3 Operating Systems

Deitel, H.M.: An Introduction to Operating Systems. Addison-Wesley, Reading (1990)

- The revised first edition includes case studies of UNIX, VMS, CP/M, MVS and VM. The second edition adds OS/2 and the Macintosh operating systems. There is a coverage of hardware, software, firmware, process management, process concepts, asynchronous concurrent processes, concurrent programming, deadlock and indefinite postponement, storage management, real storage, virtual storage, processor management, distributed computing, disk performance optimisation, file and database systems, performance, coprocessors, risc, data flow, analytic modelling, networks, security and it concludes with case studies of the these operating systems. The book is well written and an easy read.

32.7.4 Java and IEEE 754

http://www.cs.berkeley.edu/~darcy/Borneo/

- Borneo Language Homepage: Borneo is a dialect of the Java language designed to have true support for the IEEE 754 floating point standard. The status of arithmetic in Java is fluid. At the time of writing this book Sun had withdrawn from the formal language standardisation process. Sun have a publication at their web site that addresses changes to the Java language specification for JDK Release 1.2 floating point arithmetic. Their home Java page is
  - http://www.java.sun.com/

32.7.5 C and IEEE 754

http://wwwold.dkuug.dk/JTC1/SC22/WG14/

- The official home of JTC1/SC22/WG14 – C. The C programming language standard ISO/IEC 9899 was adopted by ISO in 1990. ANSI then replaced their first standard X3.159 by the ANSI/ISO 9899 standard identical to ISO/IEC 9899:1990.

Chapter 33
Miscellaneous Features and Examples

The Analytical Engine weaves algebraic patterns, just as the Jacquard loom weaves flowers and leaves.

Ada Lovelace

Aims

The aims of this chapter are to look at some additional features of Fortran and also provide coverage of a small number of other areas including

- keyword and optional arguments
- Non recursive quicksort
- Simple graphics programming – dislin

33.1 Introduction

This chapter looks at a small number of additional examples that don't really fit anywhere else. We also cover a small number of additional Fortran concepts.

33.2 Keyword and Optional Arguments

The examples of procedures so far have assumed that the dummy arguments and the corresponding arguments are in the same position, i.e., we are using positional arguments. Fortran also provides the ability to supply the actual arguments to a procedure by keyword, and hence in any order.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_33, © Springer-Verlag London Limited 2012

To do this the name of the dummy argument is referred to as the keyword and is specified in the actual argument list in the form

```
dummy-argument = actual-argument
```

To illustrate this, let us consider a subroutine to solve ordinary differential equations. The full subroutine and explanation are given in Chap. 25:

```
subroutine Runge_Kutta_Merson(Y,FUN,IFAIL,N,A,B,TOL)
```

where A is the initial point, B is the end point at which the solution is required, TOL is the accuracy to which the solution is required and N is the number of equations.

The subroutine can be called as follows:

```
call Runge_Kutta_Merson ( Y , Fun1 , IFAIL , A=0.0 ,&
   B=8.0 , Tol=1.0E-6 , N=3)
```

where the dummy arguments A, B, Tol and N are now being used as keywords. The use of keyword arguments makes the code easier to read and decreases the need to remember their precise position in the argument list.

Also with Fortran comes the ability to specify that an argument is optional. This is very useful when designing procedures for use by a range of programmers. Inside a procedure defaults can be set for the optional arguments providing an easy-to-use interface, while at the same time allowing sophisticated users a more comprehensive one.

To declare a dummy argument to be optional the optional attribute can be used. For example, the last dummy argument Tol for the subroutine Runge_Kutta_Merson could be declared to be optional (although internally in the subroutine the code would have to be changed to allow for this), e.g.,

```
subroutine Runge_Kutta_Merson(Y,FUN,IFAIL,N,A,B,Tol)
use Precision_module
   real(Long), intent(inout), optional :: Tol
```

and because it is at the end of the dummy argument list, calling the subroutine with a positional argument list, Tol can be omitted, e.g.,

```
call Runge_Kutta_Merson(Y,Fun1,IFAIL,N,A,B)
```

The code of the subroutine will need to be changed to check to see if the argument Tol is supplied, the intrinsic function PRESENT being available for this purpose. Sample code is given below:

```
subroutine Runge_Kutta_Merson(Y,FUN, IFAIL, N,A,B,Tol)
use Precision_module
! code left out
real(Long),intent(in),optional::Tol
real(Long)::Internal_tol = 1.0D-3
  if(PRESENT(Tol)) then
    Internal_tol=Tol
    print*, 'Tol = ', Internal_tol,' is supplied'
```

```
  else
    print*,"Tol isn't supplied, default tolerance = "
    print *,Internal_tol,' is used'
  endif
! code left out but all references to tol
! would have to be changed to internal_tol
end subroutine Runge_Kutta_Merson
```

A number of points need to be noted when using keyword and optional arguments:

- if all the actual arguments use keywords, they may appear in any order.
- When only some of the actual arguments use keywords, the first part of the list must be positional followed by keyword arguments in any order.
- When using a mixture of positional and keyword arguments, once a keyword argument is used all subsequent arguments must be specified by keyword.
- if an actual argument is omitted the corresponding optional dummy argument must not be redefined or referenced, except as an argument to the PRESENT intrinsic function.
- if an optional dummy argument is at the end of the argument list then it can just be omitted from the actual argument list.
- Keyword arguments are needed when an optional argument not at the end of an argument list is omitted, unless all the remaining arguments are omitted as well.
- Keyword and optional arguments require explicit procedure interfaces, i.e., the procedure must be internal, a module procedure or have an interface block available in the calling program unit.

A number of the intrinsic procedures we have used have optional arguments. Consult Appendix C for details.

33.3 Allocatable Dummy Arrays

In the recursive subroutine example using quicksort in Chap. 20 the allocation took place in the main program. In this example the allocation takes place in the read_data subroutine.

```
module read_data_module
  implicit none

contains
  subroutine read_data(file_name,raw_data,how_many)
    implicit none
    character (len=*), intent (in) :: file_name
    integer, intent (in) :: how_many
    real, intent (out), allocatable, dimension (:) ::
raw_data
```

```
! local variables
    integer :: i
    allocate(raw_data(1:how_many))
    open (file=file_name,unit=1)
    do i = 1, how_many
      read (unit=1,fmt=*) raw_data(i)
    end do

  end subroutine read_data
end module read_data_module

module sort_data_module
   implicit none

contains
  subroutine sort_data(raw_data,how_many)
    implicit none
    integer, intent (in) :: how_many
    real, intent (inout), dimension (:) :: raw_data

    call quicksort(1,how_many)

  contains

    recursive subroutine quicksort(l,r)
       implicit none
       integer, intent (in) :: l, r
! local variables
       integer :: i, j
       real :: v, t

       I = l
       j = r
       v = raw_data(int((l+r)/2))
       do
         do while (raw_data(i)<v)
           I = I + 1
         end do
         do while (v<raw_data(j))
           j = j - 1
         end do
         if (i<=j) then
           t = raw_data(i)
           raw_data(i) = raw_data(j)
           raw_data(j) = t
           I = I + 1
           j = j - 1
         end if
```

```
        if (i>j) exit
      end do

      if (l<j) then
        call quicksort(l,j)
      end if

      if (i<r) then
        call quicksort(i,r)
      end if
    end subroutine quicksort

  end subroutine sort_data
end module sort_data_module

module print_data_module
    implicit none

contains
  subroutine print_data(raw_data,how_many)
    implicit none
    integer, intent (in) :: how_many
    real, intent (in), dimension (:) :: raw_data
! local variables
    integer :: i

    open (file='sorted.txt',unit=2)
    do i = 1, how_many
      write (unit=2,fmt=*) raw_data(i)
    end do
    close (2)
  end subroutine print_data

end module print_data_module

program ch3301
  use read_data_module
  use sort_data_module
  use print_data_module

  implicit none
  integer :: how_many
  character (len=20) :: file_name
  real, allocatable, dimension (:) :: raw_data
  integer, dimension (8) :: timing

  print *, ' how many data items are there?'
  read *, how_many
```

```
  print *, ' what is the file name?'
  read '(a)', file_name
  call date_and_time(values=timing)
  print *, ' initial'
  print *, timing(6), timing(7), timing(8)
  call read_data(file_name,raw_data,how_many)
  call date_and_time(values=timing)
  print *, ' allocate and read'
  print *, timing(6), timing(7), timing(8)
  call sort_data(raw_data,how_many)
  call date_and_time(values=timing)
  print *, ' sort'
  print *, timing(6), timing(7), timing(8)
  call print_data(raw_data,how_many)
  call date_and_time(values=timing)
  print *, ' print'
  print *, timing(6), timing(7), timing(8)
  print *, ' '
  print *, ' data written to file sorted.txt'

end program ch3301
```

We now have a choice of where we do the allocation. This is more flexible than having to do the allocation in the main program, which is effectively a more Fortran 77 style of programming.

33.4 Non Recursive Quicksort

The following subroutine is a non recursive Fortran 77 implementation of quicksort. It is taken from the Netlib site. Their web address is

http://www.netlib.org/

The following is taken directly from their FAQ.

> What is Netlib? The Netlib repository contains freely available software, documents, and databases of interest to the numerical, scientific computing, and other communities. The repository is maintained by AT&T Bell Laboratories, the University of Tennessee and Oak Ridge National Laboratory, and by colleagues world-wide. The collection is replicated at several sites around the world, automatically synchronized, to provide reliable and network efficient service to the global community.

We located the subroutine by doing a search at the Netlib site on sort. One of the entries returned is to the routine dsort.f

Here is this subroutine.

```
*DECK DSORT
       SUBROUTINE DSORT (DX, DY, N, KFLAG)
C***BEGIN PROLOGUE  DSORT
C***PURPOSE  Sort an array and optionally make the same
interchanges in
C              an auxiliary array. The array may be
sorted in increasing
C              or decreasing order. A slightly
modified QUICKSORT
C              algorithm is used.
C***LIBRARY   SLATEC
C***CATEGORY  N6A2B
C***TYPE       DOUBLE PRECISION (SSORT-S, DSORT-D,
ISORT-I)
C***KEYWORDS SINGLETON QUICKSORT, SORT, SORTING
C***AUTHOR Jones, R. E., (SNLA)
C            Wisniewski, J. A., (SNLA)
C***DESCRIPTION
C
C   DSORT sorts array DX and optionally makes the same
interchanges in
C   array DY. The array DX may be sorted in
increasing order or
C   decreasing order. A slightly modified quicksort
algorithm is used.
C
C   Description of Parameters
C      DX - array of values to be sorted    (usually
abscissas)
C      DY - array to be (optionally) carried along
C      N  - number of values in array DX to be sorted
C      KFLAG - control parameter
C             =  2 means sort DX in increasing order
and carry DY along.
C             =  1 means sort DX in increasing order
(ignoring DY)
C             = -1 means sort DX in decreasing order
(ignoring DY)
C             = -2 means sort DX in decreasing order
and carry DY along.
C
C***REFERENCES R. C. Singleton, Algorithm 347, An
efficient algorithm
```

```
C                   for sorting with minimal storage,
Communications of
C                   the ACM, 12, 3 (1969), pp.
185-187.
C***ROUTINES CALLED XERMSG
C***REVISION HISTORY (YYMMDD)
C    761101  DATE WRITTEN
C    761118  Modified to use the Singleton quicksort
algorithm.  (JAW)
C    890531  Changed all specific intrinsics to
generic.  (WRB)
C    890831  Modified array declarations.   (WRB)
C    891009  Removed unreferenced statement labels.
(WRB)
C    891024  Changed category.  (WRB)
C    891024  REVISION DATE from Version 3.2
C    891214  Prologue converted to Version 4.0 format.
(BAB)
C    900315  CALLs to XERROR changed to CALLs to
XERMSG.   (THJ)
C    901012  Declared all variables; changed X,Y to
DX,DY; changed
C              code to parallel SSORT. (M. McClain)
C    920501  Reformatted the REFERENCES section. (DWL,
WRB)
C    920519  Clarified error messages.   (DWL)
C    920801  Declarations section rebuilt and code
restructured to use
C             IF-THEN-ELSE-ENDIF.  (RWC, WRB)
C***END PROLOGUE  DSORT
C      .. Scalar Arguments ..
       INTEGER KFLAG, N
C      .. Array Arguments ..
       DOUBLE PRECISION DX(*), DY(*)
C      .. Local Scalars ..
       DOUBLE PRECISION R, T, TT, TTY, TY
       INTEGER I, IJ, J, K, KK, L, M, NN
C      .. Local Arrays ..
       INTEGER IL(21), IU(21)
C      .. External Subroutines ..
       EXTERNAL XERMSG
C      .. Intrinsic Functions ..
```

```
       INTRINSIC ABS, INT
C***FIRST EXECUTABLE STATEMENT DSORT
       NN = N
       IF (NN .LT. 1) THEN
          CALL XERMSG ('SLATEC', 'DSORT'
     +       'The number of values to be sorted is not
positive.', 1, 1)
          RETURN
      ENDIF
C
      KK = ABS(KFLAG)
      IF (KK.NE.1 .AND. KK.NE.2) THEN
          CALL XERMSG ('SLATEC', 'DSORT',
     +      'The sort control parameter, K, is not 2,
1, -1, or -2.', 2,
     +       1)
          RETURN
      ENDIF
C
C     Alter array DX to get decreasing order if needed
C
      IF (KFLAG .LE. -1) THEN
          DO 10 I=1,NN
             DX(i) = -DX(i)
   10    CONTINUE
      ENDIF
C
      IF (KK .EQ. 2) GO TO 100
C
C     Sort DX only
C
      M = 1
      I = 1
      J = NN
      R = 0.375D0
C
   20 IF (I .EQ. J) GO TO 60
      IF (R .LE. 0.5898437D0) THEN
          R = R+3.90625D-2
      ELSE
          R = R-0.21875D0
      ENDIF
```

```
C
   30 K = I
C
C      Select a central element of the array and save
it in location T
C
       IJ = I + INT((J-I)*R)
       T = DX(IJ)
C
C      If first element of array is greater than T,
interchange with T
C
       IF (DX(i) .GT. T) THEN
           DX(IJ) = DX(i)
           DX(i) = T
           T = DX(IJ)
       ENDIF
       L = J
C
C      If last element of array is less than T,
interchange with T
C
       IF (DX(J) .LT. T) THEN
           DX(IJ) = DX(J)
           DX(J) = T
           T = DX(IJ)
C
C          If first element of array is greater than T,
interchange with T
C
           IF (DX(i) .GT. T) THEN
               DX(IJ) = DX(i)
               DX(i) = T
               T = DX(IJ)
           ENDIF
       ENDIF
C
C      Find an element in the second half of the array
which is smaller
C      than T
C
    40 L = L-1
```

```
       IF (DX(L) .GT. T) GO TO 40
C
C      Find an element in the first half of the array
which is greater
C      than T
C
    50 K = K+1
       IF (DX(K) .LT. T) GO TO 50
C
C      Interchange these elements
C
       IF (K .LE. L) THEN
          TT = DX(L)
          DX(L) = DX(K)
          DX(K) = TT
          GO TO 40
       ENDIF
C
C      Save upper and lower subscripts of the array yet
to be sorted
C
       IF (L-I .GT. J-K) THEN
          IL(M) = I
          IU(M) = L
          I = K
          M = M+1
       ELSE
          IL(M) = K
          IU(M) = J
          J = L
          M = M+1
       ENDIF
       GO TO 70
C
C      Begin again on another portion of the unsorted
array
C
    60 M = -1
       IF (M .EQ. 0) GO TO 190
       I = IL(M)
       J = IU(M)
```

```
C
   70 IF (J-I .GE. 1) GO TO 30
      IF (I .EQ. 1) GO TO 20
      I = I-1
C
   80 I = I+1
      IF (I .EQ. J) GO TO 60
      T = DX(I+1)
      IF (DX(i) .LE. T) GO TO 80
      K = I
C
   90 DX(K+1) = DX(K)
      K = K-1
      IF (T .LT. DX(K)) GO TO 90
      DX(K+1) = T
      GO TO 80
C
C     Sort DX and carry DY along
C
  100 M = 1
      I = 1
      J = NN
      R = 0.375D0
C
  110 IF (I .EQ. J) GO TO 150
      IF (R .LE. 0.5898437D0) THEN
         R = R+3.90625D-2
      ELSE
         R = R-0.21875D0
      ENDIF
C
  120 K = I
C
C     Select a central element of the array and save
it in location T
C
      IJ = I + INT((J-I)*R)
      T = DX(IJ)
      TY = DY(IJ)
C
C     If first element of array is greater than T,
interchange with T
```

```
C
       IF (DX(I) .GT. T) THEN
           DX(IJ) = DX(I)
           DX(i) = T
           T = DX(IJ)
           DY(IJ) = DY(I)
           DY(i) = TY
           TY = DY(IJ)
       ENDIF
       L = J
C
C      If last element of array is less than T,
interchange with T
C
       IF (DX(J) .LT. T) THEN
           DX(IJ) = DX(J)
           DX(J) = T
           T = DX(IJ)
           DY(IJ) = DY(J)
           DY(J) = TY
           TY = DY(IJ)
C
C          If first element of array is greater than T,
interchange with T
C
           IF (DX(I) .GT. T) THEN
               DX(IJ) = DX(i)
               DX(I) = T
               T = DX(IJ)
               DY(IJ) = DY(I)
               DY(I) = TY
               TY = DY(IJ)
           ENDIF
       ENDIF
C
C      Find an element in the second half of the array
which is smaller
C      than T
C
  130 L = L-1
      IF (DX(L) .GT. T) GO TO 130
```

```
C
C      Find an element in the first half of the array
which is greater
C      than T
C
  140 K = K+1
      IF (DX(K) .LT. T) GO TO 140
C
C      Interchange these elements
C
       IF (K .LE. L) THEN
           TT = DX(L)
           DX(L) = DX(K)
           DX(K) = TT
           TTY = DY(L)
           DY(L) = DY(K)
           DY(K) = TTY
           GO TO 130
       ENDIF
C
C      Save upper and lower subscripts of the array yet
to be sorted
C
       IF (L-I .GT. J-K) THEN
           IL(M) = I
           IU(M) = L
           I = K
           M = M+1
       ELSE
           IL(M) = K
           IU(M) = J
           J = L
           M = M+1
       ENDIF
       GO TO 160
C
C      Begin again on another portion of the unsorted
array
C
  150 M = -1
       IF (M .EQ. 0) GO TO 190
       I = IL(M)
       J = IU(M)
```

```
C
  160 IF (J-I .GE. 1) GO TO 120
      IF (I .EQ. 1) GO TO 110
      I = I-1
C
  170 I = I+1
      IF (I .EQ. J) GO TO 150
      T = DX(I+1)
      TY = DY(I+1)
      IF (DX(i) .LE. T) GO TO 170
      K = I
C
  180 DX(K+1) = DX(K)
      DY(K+1) = DY(K)
      K = K-1
      IF (T .LT. DX(K)) GO TO 180
      DX(K+1) = T
      DY(K+1) = TY
      GO TO 170
C
C     Clean up
C
  190 IF (KFLAG .LE. -1) THEN
         DO 200 I=1,NN
            DX(i) = -DX(i)
  200    CONTINUE
      ENDIF
      RETURN
      END
```

We will look at the ways that we can replace our call to quicksort in ch2005 with a call to dsort. Here is the header for the dsort routine.

```
          SUBROUTINE DSORT (DX, DY, N, KFLAG)
```

The routine takes 4 parameters and we look at the implementation of the dsort routine to find out more details about each parameter. This line

```
C***TYPE         DOUBLE PRECISION (SSORT-S, DSORT-D,
ISORT-I)
```

provides the first clue as to the nature of the parameters.

The following provide some more.

```
C   Description of Parameters
C       DX - array of values to be sorted    (usually
abscissas)
C       DY - array to be (optionally) carried along
C       N  - number of values in array DX to be sorted
C       KFLAG - control parameter
C               =  2  means sort DX in increasing order
and carry DY along.
C               =  1  means sort DX in increasing order
(ignoring DY)
C               = -1  means sort DX in decreasing order
(ignoring DY)
C               = -2  means sort DX in decreasing order
and carry DY along.
```

The following lines then complete the information.

```
C       .. Scalar Arguments ..
        INTEGER KFLAG, N
C       .. Array Arguments ..
        DOUBLE PRECISION DX(*), DY(*)
```

So we have a type mismatch between the array argument to our `quicksort` and the netlib `dsort`. All we need to do here is a global search and replace of `double precision` with `real` in `dsort.f` in our favourite editor.

The second problem is what to do about the second argument the `DY` array. One solution to this problem is to use the same `raw_data` array and set `KFLAG` to 1. This ignores the `DY` array.

The next problem we have are the calls to the external routines shown below.

```
       IF (NN .LT. 1) THEN
          CALL XERMSG ('SLATEC', 'DSORT',
     +       'The number of values to be sorted is not
positive.', 1, 1)
          RETURN
       ENDIF
```

and

```
       IF (KK.NE.1 .AND. KK.NE.2) THEN
          CALL XERMSG ('SLATEC', 'DSORT',
     +       'The sort control parameter, K, is not 2,
1, -1, or -2.', 2,
     +       1)
          RETURN
       ENDIF
```

The simple solution here is just to comment out the calls to XERMSG as we know the errors cannot occur. We also need to comment out the external statement referencing XERMSG.

The following lines

```
C***REFERENCES R. C. Singleton, Algorithm 347, An
efficient algorithm

C                   for sorting with minimal storage,
Communications of
C                   the ACM, 12, 3 (1969), pp.
185-187.
C***ROUTINES CALLED  XERMSG
C***REVISION HISTORY  (YYMMDD)
C    761101  DATE WRITTEN
C    761118  Modified to use the Singleton quicksort
algorithm.  (JAW)
C    890531  Changed all specific intrinsics to
generic.  (WRB)
C    890831  Modified array declarations.   (WRB)
C    891009  Removed unreferenced statement labels.
(WRB)
C    891024  Changed category.  (WRB)
C    891024  REVISION DATE from Version 3.2
C    891214  Prologue converted to Version 4.0 format.
(BAB)
C    900315  CALLs to XERROR changed to CALLs to
XERMSG.   (THJ)
C    901012  Declared all variables; changed X,Y to
DX,DY; changed
C             code to parallel SSORT. (M. McClain)
C    920501 Reformatted the REFERENCES section. (DWL,
WRB)
C    920519  Clarified error messages.   (DWL)
C    920801  Declarations section rebuilt and code
restructured to use
C             IF-THEN-ELSE-ENDIF.  (RWC, WRB)
```

provide details about the algorithm and its revision history. This information is extremely use when working with the subroutine.

We are now going to look at one solution to the problem of how to integrate the original program and the `dsort.f` routine. We can independently compile the dsort.f routine as a Fortran 77 style routine and generate an object file. We can then compile the program and add the object file to the compilation line.

Here are solutions using a variety of compilers.

33.4.1 Gfortran

```
gfortran -c dsort.f
gfortran ch2005.f90 dsort.o
./a.out
   how many data items are there?
10000000
   what is the file name?
random.txt
   initial
             1              6          610
   allocate
             1              6          610
   read
             1             12          839
   sort
             1             15          142
   print
             1             33          713

   data written to file sorted.txt
```

33.4.2 Intel

```
>ifort -c dsort.f
>ifort ch2005.f90 dsort.obj
>ch2005
   how many data items are there?
10000000
   what is the file name?
random.txt
   initial
            56           30           408
   allocate
            56           30           412
   read
            56           34           148
   sort
            56           35           521
   print
            57            3           179

   data written to file sorted.txt
```

33.4.3 Nag

```
>nagfor -c dsort.f
>nagfor ch2005.f90 dsort.o
>a.exe
  how many data items are there?
10000000
  what is the file name?
random.txt
  initial
 1  4  970
  allocate
 1  4  975
  read
 1  8  721
  sort
 1  11  135
  print
 1  20  378

data written to file sorted.txt
```

33.4.4 Notes – Version Control Systems

The original program had the following statement

*DECK DSORT

and this statement was one used in version control or revision control software of the time. Two that were available on CDC systems from the 1970s and 1980s were called update and modify and they used so called `deck names` as seen in this example. In computer programming, revision control is any practice that tracks and provides control over changes to source code. Software developers also use revision control software to maintain documentation and configuration files as well as source code.

The use of this kind of software is common for medium to large scale program development.

Wikipedia provides a comparison of what is currently available. See

http:/en.wikipedia.org/wiki/Comparison_of_revision_control_software

for more information.

33.5 Simple Graphics Programming – Dislin

We have already used the dislin graphics library in earlier chapters. Our resource file

http://www.fortranplus.co.uk/resources/fortran_resources.pdf

provides details of some of the graphics libraries available. This is the dislin home page.

http://www.mps.mpg.de/dislin/

Here is a description of the software from the above page.

- The software is available for several C, Fortran 77 and Fortran 90/95 compilers on the operating systems UNIX, Linux, FreeBSD, Open VMS, Windows, Mac OSX and MS-DOS. DISLIN programs are very system-independent, they can be ported from one operating system to another with out any changes.

The original program on which this is based was written by Ian whilst he was on secondment to the United Nations Environment Programme. Section 9 of their Environmental Data Reports cover natural disasters and these include

- Earthquakes
- Volcanoes
- Tsunamis
- Floods
- Landslides
- Natural dams
- Droughts
- Wildfires

See the bibliography for more details of these publications. The tsunami data sets are from this chapter.
The tsunami data file can be found at:

http://www.fortranplus.co.uk/

There are some minor wrap problems with the code:

```
PROGRAM ch3303
USE DISLIN
LOGICAL :: trial, screen
REAL :: long, lat

! Switch diagnostics  on or off. This was used
! when developing
! the original uniras version of the code.
! I haven't needed
! to switch it on with the dislin version.

  screen = .FALSE.
  trial = .FALSE.

!       PRINT *,' Testing on/off'
!       READ (UNIT=*, FMT=*, END=9,ERR=9)TRIAL
!9      CONTINUE

!      Read in the tsunami data

  CALL datain(trial)

!     I now have all the tsunami data latitude
!  and longitude
!     values read in to the arrays in the
! TSUNAM common block.


! The following options were available
!  with the original
! uniras version of the code.
! They were not available with
! the dislin version. They are therefore commented out.
!     The next thing to do is sort out what kind of
```

```
!     map projection the user wants. I am offering
!     one of the following
!     Projection
!     Lambert        - equal area        -  rectangle
!     Mercator      - equal direction   -  rectangle
!     Hammer         - equal area        -  oval
!     Bonne          -                    -  heart
!     Orthographic  - globe            -  round
!  PRINT *, ' What projection would you like?'
!  PRINT *
!  PRINT *, ' 1 = Lambert        - equal area'
!  PRINT *, '        ' - rectangle'
!  PRINT *, ' 2 = Mercator       - equal direction'
!  PRINT *, '   - rectangle'
!  PRINT *, ' 3 = Hammer         - equal area'
!  PRINT *, ' - oval'
!  PRINT *, ' 4 = Bonne          -'
!  PRINT *, '                   -  heart'
!  PRINT *, ' 5 = Orthographic -'
!  PRINT *, ' globe              -  round'
!100 READ (unit=*,fmt=*,end=110,err=110) iproj
!110 IF ((iproj<1) .OR. (iproj>5)) THEN
!     PRINT *, ' Please input a number in the range
!     PRINT *, ' 1 to 5'
!     GO TO 100
!  END IF
!     The next thing to do is set the centre
!  of the map
!     In this case the Pacific

  iproj=1
  lat = 0.0
  long = 180.0

!  The following options were available with
!  the original
!  uniras version of the code.
!  They were not available with
!  the dislin version. They are therefore commented out.
!       I offer the user a choice of region
!       to plot.
!   PRINT *, ' Which region do you wish to plot?'
```

```
!
!  PRINT *, ' 1 = Hawaii'
!  PRINT *, ' 2 = New Zealand and'
!  PRINT *, ' South Pacific Islands'
!  PRINT *, ' 3 = Papua New Guinea and'
!  PRINT *, ' Soloman Islands'
!  PRINT *, ' 4 = Indonesia'
!  PRINT *, ' 5 = Philippines'
!  PRINT *, ' 6 = Japan'
!  PRINT *, ' 7 = Kuril Islands and Kamchatka'
!
!  PRINT *, ' 9 = West Coast -'
!  PRINT *, 'North and Central America'
!  PRINT *, ' 10 = West Coast - South America'
!120 READ (unit=*,fmt=*,end=130,err=130) nreg
!130 IF ((nreg<0) .OR. (nreg>10)) THEN
!     PRINT *, ' Please input a number between'
!     PRINT *, ' 0 and 10 inclusive'
!     GO TO 120
!  END IF

  nreg=0

! dislin initialisation routines and setting
! of some basic components
! of the plot. These are based on two
! sample dislin programs.

! Initialise dislin

  CALL DISINI

! Choose font

  CALL PSFONT('Times-Roman')


! determines the position of an axis system.
! the lower left corner of the axis system

  CALL AXSPOS(400,1850)
```

```
! The size of the axis system
! are the length and height of an axis system
! in plot coordinates. The default
! values are set to 2/3 of the page length and height.

  CALL AXSLEN(2400,1400)

! Define axis title

  CALL NAME('Longitude','X')

! Define axis title

  CALL NAME('Latitude','Y')

! This routine plots a title over an axis system.

  CALL TITLIN('Plot of 3034 Tsunami events ',3)

! determines which label types will be
! plotted on an axis.
! MAP defines geographical labels which are
! plotted as non negative floating-point
! numbers with the following
! characters 'W', 'E', 'N' and 'S'.

  CALL LABELS('MAP','XY')

! plots a geographical axis system.

  CALL GRAFMP(-180.,180.,-180.,90.,-90.,90.,-90.,30.)

! The statement CALL GRIDMP (I, J) overlays
! an axis system with a longitude
! and latitude grid where I and J are
! the number of grid lines between labels in
! the X- and Y-direction.

  CALL GRIDMP(1,1)

! The routine WORLD plots coastlines and lakes.
```

```
  CALL WORLD

! The angle and height of the characters can
! be changed with the routines
! ANGLE and HEIGHT.

  CALL HEIGHT(50)

! This routine plots a title over an axis system.
! The title may contain up to four
! lines of text designated
! with TITLIN.

  CALL TITLE

! This is a call to the convert routine.
! This was required by UNIRAS
!  CALL convrt(trial)

! This is a call to the routine that
! actually plots each event.

  CALL plotem(trial,nreg)

! DISFIN terminates DISLIN and prints a message
! on the screen. The level is set back to 0.

  CALL DISFIN

END PROGRAM ch3303

SUBROUTINE datain(trial)
 COMMON /TSUNAM/ &
  reg0la(378) , &
  reg0lo(378) , &
  reg1la(206) , &
  reg1lo(206) , &
  reg2la(41) , &
  reg2lo(41) , &
  reg3la(54) , &
  reg3lo(54) , &
  reg4la(60) , &
  reg4lo(60) , &
```

```
  reg5la(1540) , &
  reg5lo(1540) , &
  reg6la(80) , &
  reg6lo(80) , &
  reg7la(144) , &
  reg7lo(144) , &
  reg8la(245) , &
  reg8lo(245) , &
  reg9la(285) , &
  reg9lo(285)

! This subroutine reads in the tsunami data
! UNIRAS uses the following unit numbers
! 5,6,7,20,21,22,24,25,26,27,28,33
! So I have used 50.

LOGICAL :: trial
CHARACTER (80) :: filnam
  IF (trial) THEN
    PRINT *, ' Entering data input phase'
  END IF
  filnam = 'tsunami.dat'
  OPEN (unit=50,file=filnam,err=100,status='OLD')
  GO TO 110
100 PRINT *, ' Error opening data file'
  PRINT *, ' Program terminates'
  STOP
110 DO I = 1, 378
    READ (unit=50,fmt=1000) reg0la(i), reg0lo(i)
  END DO
1000 FORMAT (1X,F7.2,2X,F7.2)
  DO I = 1, 206
    READ (unit=50,fmt=1000) reg1la(i), reg1lo(i)
  END DO
  DO I = 1, 41
    READ (unit=50,fmt=1000) reg2la(i), reg2lo(i)
  END DO
  DO I = 1, 54
    READ (unit=50,fmt=1000) reg3la(i), reg3lo(i)
  END DO
  DO I = 1, 60
```

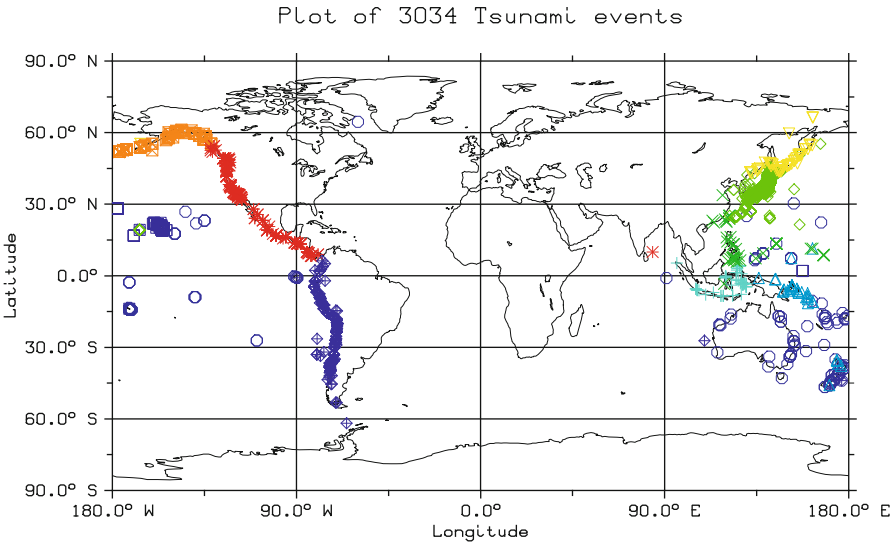
```
    READ (unit=50,fmt=1000) reg4la(i), reg4lo(i)
  END DO
  DO I = 1, 1540
    READ (unit=50,fmt=1000) reg5la(i), reg5lo(i)
  END DO
  DO I = 1, 80
    READ (unit=50,fmt=1000) reg6la(i), reg6lo(i)
  END DO
  DO I = 1, 144
    READ (unit=50,fmt=1000) reg7la(i), reg7lo(i)
  END DO
  DO I = 1, 245
    READ (unit=50,fmt=1000) reg8la(i), reg8lo(i)
  END DO
  DO I = 1, 285
    READ (unit=50,fmt=1000) reg9la(i), reg9lo(i)
  END DO
  IF (trial) THEN
    DO I = 1, 10
      PRINT *, reg0la(i), '  ', reg0lo(i)
    END DO
    PRINT *, ' Exiting data input phase'
    READ *, dummy
  END IF
END SUBROUTINE datain

SUBROUTINE plotem(trial,nreg)
USE DISLIN
 COMMON /TSUNAM/ &
  reg0la(378) , &
  reg0lo(378) , &
  reg1la(206) , &
  reg1lo(206) , &
  reg2la(41) , &
  reg2lo(41) , &
  reg3la(54) , &
  reg3lo(54) , &
  reg4la(60) , &
  reg4lo(60) , &
  reg5la(1540) , &
  reg5lo(1540) , &
  reg6la(80) , &
```

```
  reg6lo(80) , &
  reg7la(144) , &
  reg7lo(144) , &
  reg8la(245) , &
  reg8lo(245) , &
  reg9la(285) , &
  reg9lo(285)

! This subroutine plots all of the tsunamis
! onto the map as coloured
! points, with a different colour per region.
! I have chosen
! a dot size of 1 mm, and step through
! the colour pallette.
! The default may not be appropriate.

  LOGICAL :: trial
  INTEGER :: nreg
  INTEGER :: kolour=10
  DATA dwidth/1.0/
  IF (trial) THEN
    dwidth = 5.0
    PRINT *, ' Entering Plot points'
  END IF
  CALL INCMRK(-1)
  IF (nreg==0) THEN
    CALL SETCLR(kolour)
    CALL CURVMP(reg0lo,reg0la,378)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg1lo,reg1la,206)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg2lo,reg2la,41)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg3lo,reg3la,54)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg4lo,reg4la,60)
    kolour = kolour +30
    CALL SETCLR(kolour)
```

```
    CALL CURVMP(reg5lo,reg5la,1540)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg6lo,reg6la,80)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg7lo,reg7la,144)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg8lo,reg8la,245)
    kolour = kolour +30
    CALL SETCLR(kolour)
    CALL CURVMP(reg9lo,reg9la,285)
  ELSE IF (nreg==1) THEN
    kolour = 10
    CALL SETCLR(kolour)
    CALL CURVMP(reg0lo,reg0la,378)
  ELSE IF (nreg==2) THEN
    kolour = 20
    CALL SETCLR(kolour)
    CALL CURVMP(reg1lo,reg1la,206)
  ELSE IF (nreg==3) THEN
    kolour = 30
    CALL SETCLR(kolour)
    CALL CURVMP(reg2lo,reg2la,41)
  ELSE IF (nreg==4) THEN
    kolour = 40
    CALL SETCLR(kolour)
    CALL CURVMP(reg3lo,reg3la,54)
  ELSE IF (nreg==5) THEN
    kolour = 50
    CALL SETCLR(kolour)
    CALL CURVMP(reg4lo,reg4la,60)
  ELSE IF (nreg==6) THEN
    kolour = 60
    CALL SETCLR(kolour)
    CALL CURVMP(reg5lo,reg5la,1540)
  ELSE IF (nreg==7) THEN
    kolour = 70
    CALL SETCLR(kolour)
    CALL CURVMP(reg6lo,reg6la,80)
  ELSE IF (nreg==8) THEN
```

```
    kolour = 80
    CALL SETCLR(kolour)
    CALL CURVMP(reg7lo,reg7la,144)
  ELSE IF (nreg==9) THEN
    kolour = 90
    CALL SETCLR(kolour)
    CALL CURVMP(reg8lo,reg8la,245)
  ELSE IF (nreg==10) THEN
    kolour = 100
    CALL SETCLR(kolour)
    CALL CURVMP(reg9lo,reg9la,285)
  END IF
  IF (trial) THEN
    PRINT *, ' Exiting Plot points'
  END IF

END SUBROUTINE plotem
```

Here is the plot produced by this program.



As you can see there are a lot of tsunami events in the Pacific area. A colour A4 pdf of the plot can be found at the Fortranplus website.

It is a common requirement in science and engineering to have to produce graphical output and we have now briefly covered some of the capability of the dislin library. Most graphics libraries will offer similar functionality.

33.6 Problem

1. The complete working version of the non recursive version of quicksort has not been included. The source files (Netlib dsort.f and our ch2005.f90) required are available at the Fortranplus website.

Download them and make the changes necessary to replace the call to quicksort with a call to dsort with your compiler. What timing information do you get?

33.6.1 Hint

diff is a Unix command that compares text files. Here is the diff output from comparing the original dsort.f file with a working version.

```
$ diff dsort.f dsort_original.f
10c10
< C***TYPE       REAL (SSORT-S, DSORT-D, ISORT-I)
-
> C***TYPE       DOUBLE PRECISION (SSORT-S, DSORT-D,
ISORT-I)
54c54
<         REAL DX(*), DY(*)
-
>         DOUBLE PRECISION DX(*), DY(*)
56c56
<         REAL R, T, TT, TTY, TY
-
>         DOUBLE PRECISION R, T, TT, TTY, TY
61c61
< C      EXTERNAL XERMSG
-
>         EXTERNAL XERMSG
67,69c67,69
< C           CALL XERMSG ('SLATEC', 'DSORT',
< C      +       'The number of values to be sorted is
not positive.', 1, 1)
< C            RETURN
-
>             CALL XERMSG ('SLATEC', 'DSORT',
```

```
>       +         'The number of values to be sorted is
not positive.', 1, 1)
>            RETURN
74,77c74,77
< C            CALL XERMSG ('SLATEC', 'DSORT',
< C      +        'The sort control parameter, K, is not
2, 1, -1, or -2.', 2,
< C      +       1)
< C           RETURN
-
>            CALL XERMSG ('SLATEC', 'DSORT',
>       +        'The sort control parameter, K, is not
2, 1, -1, or -2.', 2,
>       +         1)
>            RETURN
```

diff is a very useful utility for comparing different versions of your programs!

33.7 Bibliography

United Nations Environment Programme: Environmental Data Report: 1989–1990, Second Edition, Blackwell Reference, Oxford (1989)

United Nations Environment Programme: Environmental Data Report: 1991–1992, Third Edition, Blackwell Reference, Oxford (1991)

Chapter 34
Converting from Fortran 77

Twas brillig, and the slithy toves
did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Lewis Carroll

Aim

This chapter looks at some of the options available when working with older Fortran code.

34.1 Introduction

This chapter looks at converting Fortran 77 code to a modern Fortran style.

The aim is to provide the Fortran 77 programmer (and in particular the person with legacy code) with some simple guidelines for conversion.

The first thing that one must have is a thorough understanding of the newer, better language features of Fortran. It is essential that the material in the earlier chapters of this book are covered, and some of the problems attempted. This will provide a feel for modern Fortran.

The second thing one must have is a thorough understanding of the language constructs used in your legacy code. Use should be made of the compiler documentation for whatever Fortran 77 compiler you are using, as this will provide the detailed (often system specific) information required. The recommendations below are therefore brief.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9_34, © Springer-Verlag London Limited 2012

It is possible to move gradually from Fortran 77 to modern Fortran. In many cases existing code can be quite simply recompiled by a suitable choice of compiler options. This enables us to mix and match old and new in one program. This process is likely to highlight nonstandard language features in your old code. There will inevitably be some problems here.

The standard identifies two kinds of decremented features; deleted and obsolescent. It is extremely unwise to consider the long-term use of these features as they are candidates for removal from future standards.

34.2 Deleted Features

The list of deleted features for Fortran 2008 is empty, i.e., there are none.

34.3 Obsolescent Features

The obsolescent features are those for which better methods are available. They are given below with alternatives.

34.3.1 Arithmetic if

Use the if statement.

34.3.2 Real and Double Precision Do Control Variables

Use integer.

34.3.3 Shared Do Termination and Non-enddo Termination

Use an end do.

34.3.4 Alternate Return

Use a case statement on return. An error code has to be returned.

34.3.5 Pause Statement

System specific. Normally easily replaced with a suitable read statement.

34.3.6 Assign and Assigned Goto Statements

Fortunately rarely used.

34.3.7 Assigned Format Statements

Use character arrays, arrays and constants.

34.3.8 H Editing

Use character edit descriptor.

34.4 Better Alternatives

Below we are looking at the new features of the Fortran standard, and how we can replace our current coding practices with the better facilities that now exist.

- Double precision – use KIND, see Chap. 5, and examples throughout the book.
- Fixed format – use free format
- Implicit typing – use implicit none
- Block data – use modules
- Common statement – use modules
- Equivalence – Invariably the use of this feature requires considerable system specific knowledge. There will be cases where there have been extremely good reasons why this feature has been used, normally efficiency related. However with the rapid changes taking place in the power and speed of hardware these reasons are diminishing.
- Assumed-size/explicit-shape dummy array arguments – if a dummy argument is assumed-size or explicit-shape (the only ones available in Fortran 77) then the ranks of the actual argument and the associated argument don't have to be the same. With Fortran arrays are now objects instead of a linear sequence of elements, as was the case with Fortran 77, and now for array arguments the fundamental rule is that actual and dummy arguments have the same rank and same

extents in each dimension, i.e., the same shape, and this is done using assumed-shape dummy array arguments. An explicit interface is mandatory for assumed-shape arrays.
- Entry statement – use module plus use statement.
- Statement functions – use internal function, see Chap. 12.
- Computed goto – use case statement, see Chap. 13.
- Alternate return – use error flags on calling routine.
- external statement for dummy procedure arguments – use modules and interface blocks. See the Runge-Kutta-Merson example in Chap. 25.

Use explicit interfaces everywhere, i.e. use module procedures. This also provides argument checking and other benefits.

34.5 Commercial Conversion Tools

At the time of writing there are several options. Have a look at our Fortran resource file:

http://www.fortranplus.co.uk/resources/fortran_resources.pdf

for up to date information.

Here are brief details of the tools currently available.

34.5.1 Convert

Fortran 77 to Fortran 90 converter by Mike Metcalf.

http://www.nag.co.uk/nagware/Examples/convert.f90

34.5.2 Forcheck

A Fortran analyzer and programming aid.

http://www.forcheck.nl/

34.5.3 Forstruct

Restructures FORTRAN into Clean, Maintainable Code.

http://www.cobalt-blue.com/fs/fsmain.htm

34.5.4 Forstudy

Analyzes and Documents your FORTRAN code.

http://www.cobalt-blue.com/

34.5.5 Fortran90-Lint

For Fortran 90 program analysis

http://legacy.cleanscape.net/products/downloads/ftpflint.html

34.5.6 Plusfort

Fortran 77 to Fortran 90 converter.

http://www.polyhedron.com/

34.5.7 VAST/77to90

Fortran 77 to Fortran 90 translator

http://www.crescentbaysoftware.com/vast_77to90.html

34.6 Example of plusFORT Capability from Polyhedron Software

Below is an example from their site that looks at the same subroutine in Fortran 66, 77 and 90 styles.

34.6.1 Original Fortran 66

This subroutine picks off digits from an integer and branches depending on their value.

```
      subroutine OBACT(TODO)
      integer TODO,DONE,IP,BASE
      common /EG1/N,L,DONE
      parameter (BASE=10)
   13 if(TODO.EQ.0) GO TO 12
      I=MOD(TODO,BASE)
      TODO=TODO/BASE
      GO TO(62,42,43,62,404,45,62,62,62),I
      GO TO 13
   42 call COPY
      GO TO 127
   43 call MOVE
      GO TO 144
  404 N=-N
   44 call DELETE
      GO TO 127
   45 call print
      GO TO 144
   62 call BADACT(I)
      GO TO 12
  127 L=L+N
  144 DONE=DONE+1
      call RESYNC
      GO TO 13
   12 return
      end
```

34.6.2 Fortran 77 Version

In addition to restructuring, SPAG has renamed some variables, removed the unused variable IP, inserted declarations, and used upper and lower case to distinguish different types of variable:

```
       subroutine OBACT(Todo)
       implicit none
C*** Start of declarations inserted by SPAG
       integer act , LENgth , NCHar
C*** End of declarations inserted by SPAG
       integer Todo , DONe , BASE
       common /EG1   / NCHar , LENgth , DONe
       parameter (BASE=10)
 100   if ( Todo.NE.0 ) then
          act = MOD(Todo,BASE)
          Todo = Todo/BASE
          if ( act.EQ.1 .OR. act.EQ.4 .OR.
      &        act.EQ.7 .OR. act.EQ.8 .OR.
      &        act.EQ.9 ) then
             call BADACT(act)
             goto 200
          ELSEIF ( act.EQ.2 ) then
             call COPY
             LENgth = LENgth + NCHar
          ELSEIF ( act.EQ.3 ) then
             call MOVE
          ELSEIF ( act.EQ.5 ) then
             NCHar = -NCHar
             call DELETE
             LENgth = LENgth + NCHar
          ELSEIF ( act.EQ.6 ) then
             call print
          else
             goto 100
          endif
          DONe = DONe + 1
          call RESYNC
          goto 100
       endif
 200   return
       end
```

34.6.3 Fortran 90 Version

SPAG has used do while, select case, exit and cycle. No GOTOs or labels remain.

```
      subroutine OBACT(Todo)
      implicit none
C*** Start of declarations inserted by SPAG
      integer act , LENgth , NCHar
C*** End of declarations inserted by SPAG
      integer Todo , DONe , BASE
      common /EG1    / NCHar , LENgth , DONe
      parameter (BASE=10)
      do while ( Todo.NE.0 )
         act = MOD(Todo,BASE)
         Todo = Todo/BASE
         select case (act)
         case (1,4,7,8,9)
            call BADACT(act)
            exit
         case (2)
            call COPY
            LENgth = LENgth + NCHar
         case (3)
            call MOVE
         case (5)
            NCHar = -NCHar
            call DELETE
            LENgth = LENgth + NCHar
         case (6)
            call print
         case DEFAULT
            cycle
         end select
         DONe = DONe + 1
         call RESYNC
      enddo
      return
      end
```

This tool suite can also be used in the maintenance of code during development.

34.7 Summary

This chapter has shown some of the options open to you when working with legacy code. The emphasis has been on relatively straightforward code restructuring. The use of software tools to aid in this is highly recommended as converting manually using an editor is obviously going to involve much more work.

Appendix A

Glossary

Abstract interface Set of procedure characteristics with dummy argument names

Actual argument A value (variable, expression or procedure) passed from a calling program unit to a subprogram unit.

Adjustable array An explicit-shape array that is a dummy argument to a subprogram.

Algorithm Derived from the name of the ninth century Persian mathematician Abu Ja'far Mohammed ibn Musa al-Kuwarizmi (father of Ja'far Mohammed, son of Moses, native of Kuwarizmi), corrupted through western culture as Al-Kuwarizmi. Now a sequence of computations.

Allocatable Having the allocatable attribute

Allocatable array An array that has the allocatable attribute.

Argument Exists in two forms; actual argument, which is in the calling routine and is one of a variable, expression or procedure, and dummy argument, which is in the called routine.

Argument association The process of matching up an actual argument and dummy argument during program execution.

Array An array is a data structure where each scalar element has the same type and kind. An array may be up to rank 7. It may be referenced by element (via subscripts), by section or as a whole.

Array constructor A mechanism used to initialise or give values to a one-dimensional array. The RESHAPE function can then be used to handle rank 2 and above arrays.

Array element A scalar item of an array. An array element is picked out by a subscript.

Array element ordering The elements of an array, regardless of rank, form a linear sequence. The sequence is such that the subscripts along the first dimension vary most rapidly.

Array section A part of an array. The actual set depends on the subscripts.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9, © Springer-Verlag London Limited 2012

ASCII American Standard Code for Information Interchange. See Appendix 3.

Associate name Name of construct entity associated with a selector of an associate or select type construct

Association The means by which an entity can be referenced by different names in one scoping unit, or one or more names in multiple scoping units.

Assumed-length dummy argument A dummy argument that inherits the length attribute of the actual argument.

Assumed-shape array A dummy argument that inherits the shape of the associated argument.

Assumed-size array A dummy array whose size is inherited from the associated actual argument.

Atomic subroutine Intrinsic subroutine that performs an action on its atom argument atomically

Attribute A property of a data type, and specified in a type declaration statement.

Automatic array This is an explicit-shape array that is a local variable in a subprogram unit.

Binding Type-bound procedure or final subroutine

Binding name Name given to a specific or generic type-bound procedure in the type definition

Block Sequence of executable constructs formed by the syntactic class block and which is treated as a unit

Bound The bounds of an array are the upper and lower limits of the index in each dimension.

Character constant A constant that is a string of one or more printable ASCII characters, enclosed in apostrophes (') or quotation mark (").

Character string A sequence of one or more characters. These are contiguous.

Coarray Data entity that has nonzero corank

Cobound Bound of a codimension

Codimension Dimension of the pattern formed by a set of corresponding coarrays

Collating sequence The order in which a set of characters is sorted by default. The standard does not require that a processor provide the ASCII encoding, but does require intrinsic functions that convert between the processor encoding and the ASCII encoding.

Compilation unit One or more source files that are compiled to form one object file.

Component Part of a derived type definition.

Concatenate Join together two or more character items using the character concatenation operator //.

Conformable Two arrays are said to be conformable if they have the same shape.

Constant A constant is a data object whose value cannot be changed. There are two kinds in Fortran: one is obtained using the parameter statement; the other is a literal constant in an expression; e.g., with the expression 4*ATAN(1) both 4 and 1 are literal constants. It may be a scalar or an array.

Contiguous Normally applied to items that are adjacent in memory, e.g., characters in a character variable.

Corank Number of codimensions of a coarray

Cosubcript Scalar integer expression

Data entity A data object that has a specific type.

Data object A data object is a constant, variable or part of a constant or variable.

Data type For each data type there are the following: 0. a name 1. a set of values from a domain; 2. a set of valid operations upon these values; 3. a display method. There are five predefined data types in Fortran and these are integer, real, complex, character and logical.

For integers the values are drawn from the domain of integer numbers, the valid operations are addition, subtraction, multiplication, division and exponentiation, and they are displayed as a sequence of digits.

Declaration A declaration is a nonexecutable statement that specifies attributes of a program element, e.g., specifying the dimension of an array and the type of a variable.

Default initialization Mechanism for automatically initializing pointer components to have a defined pointer association status, and nonpointer components to have a particular value.

Default kind The kind type parameter which is used for one of Fortran's base types (integer, real, complex, character or logical) if one is not specified.

Deferred-shape array An allocatable array or an array pointer. The bounds are specified with a colon, (:).

Defined For a data object having a valid value.

Defined assignment Assignment defined by a procedure

Derived type A data type that is user defined and not one of the five intrinsic types.

Dimension An array can be from one to seven dimensioned inclusive. Also called the rank.

Disassociated (Pointer association) Pointer association status of not being associated with any target and not being undefined.

Dummy argument A variable name that appears in the bracketed or parenthesised list following the procedure name. (e.g., function or subroutine name). Dummy arguments take on the actual values of the corresponding arguments in the calling routine.

Dynamic type Type of a data entity at a particular point during execution of a program

Elemental An operation that applies independently to each element in an array.

Elemental assignment Assignment that operates elementally.

Elemental procedure Elemental intrinsic procedure or procedure defined by an elemental subprogram

Entity Rather vague term covering a constant, variable, program unit, etc.

Exceptional values Normally restricted to real numbers and typically one of nonnormalised numbers, infinity, not-a-number (NaN) values, etc.

Explicit interface A mechanism to make information available between the calling routine and the called routine. This information includes the names of the procedures, the dummy arguments, the attributes of the arguments, the attributes of functions, and the order of the arguments.

Explicit-shape array A named array that has its bounds specified in each dimension.

Expression An expression is a sequence of operands and operators that specifies a computation.

Extended type Type with the extends attribute

Extent The number of elements of one dimension of an array. Also called the size.

External subprogram An external subprogram is one that is global to the whole program.

Function One of the two procedure mechanisms available in Fortran along with the subroutine. They effectively provide a way of invoking a computation by using the function name, and return a result. There is the concept of type and kind for the result.

Function reference A function is invoked by the use of its name in an expression.

Function result The result value(s) from invoking a function.

Generic Simplistically the ability of a procedure to accept arguments of more than type. This facility is taken for granted with the intrinsic procedures, and users can now create their own generic procedures.

Global An entity that is available throughout the executable program. A global entity has global scope. See also scope and local scope.

Host association The mechanism by which a module procedure, internal procedure or derived type definition accesses entities of the host.

Image Instance of a Fortran program

Image control statement Statement that affects the execution ordering between images

Image index Integer value identifying an image

Implicit interface A procedure interface whose properties are not known within the scope of the calling routine.

Inherit (Extended type) acquire entities through type extension from the parent type

Inheritance association Association between the inherited components of an extended type and the components of its parent component

Inquiry function A function whose result depends on the properties of the argument.

Interface (Procedure) name, procedure characteristics, dummy argument names, binding label, and generic identifiers

Interface block A sequence of statements starting with an interface statement and ending with an end interface statement.

Interface body The sequence of statements in an interface block between either a function or subroutine statement and the corresponding end statement.

Internal procedure A procedure that is contained within an internal subprogram. The program unit containing the internal procedure is called the host. The internal procedure is local to the host and inherits the host environment through host association.

Intrinsic procedure One of the standard supplied procedures.

Keyword Statement keyword, argument keyword, type parameter keyword, or component keyword

Kind For each of the five Fortran types (integer, real, complex, logical and character) there is the concept of kind. For example for integers it is common to find 8-bit, 16-bit and 32-bit implementations. Each of these has an associated kind type.

For real and complex types this enables us to choose both the range and precision of the numbers we work with.

For characters we can choose between character sets, which is of considerable use for working with different languages.

Kind type parameter An integer value used to identify the kind of one of the five base types, see above.

Language extension Most compiler implementations will provide language extensions. These are NOT part of the standard, and make porting code suites between different hardware and software platforms difficult and sometimes impossible.

Linker A program that is normally the final stage in the process of going from Fortran source to executable.

Local entity An entity that is only available within the context of a subprogram.

Main program A program unit that contains a program statement.

Module A program unit that contains specifications and definitions that other program units can access and use.

Module procedure A function or subroutine defined within a module

Name An entity with a program, e.g., constant, variable, function result, procedure, program unit, dummy argument.

Name association This provides access to the same entity (either data or a procedure) from different scoping units by the same or a different name.

Nesting The placing of one entity within another, e.g., one loop within another or one subprogram within another.

Nonexecutable statement A language statement that describes program attributes, but does not cause any action when the program is executed.

Object file File created after successful compilation. Used by the linker to generate an executable.

Parameter Term used to describe two completely different things. 1. a named constant—and hence the parameter attribute. 2. more generally equivalent to argument.

Parent type (Extended type) Type named in the extends clause

Pointer A data object that has the pointer attribute.

Pointer assignment Association of a pointer with a target, by execution of a pointer assignment statement or an intrinsic assignment statement for a derived-type object that has the pointer as a subobject

Pointer association The association of a part of memory to a pointer by means of a target.

Precision The number of significant digits in a real number.

Procedure A function or subroutine.

Procedure interface The statements that specify the name of a procedure, the characteristics of that procedure, the name of the dummy arguments, the attributes of the dummy arguments the generic identifier (optional) for the procedure.

Program A program is an entity that can be compiled and executed on its own. There must be at least a declaration block and execution block.

Program unit A main program or a subprogram. The subprogram can be a function, subroutine or module.

Rank The rank of an array is the number of dimensions.

Recursion A property of a function or subroutine, and it means that the function or subroutine references itself directly or indirectly.

Reference A data object reference is the appearance of a named entity in an executable statement requiring the value of the object.

Relational expression An expression containing one or more of the relational operators and operands of numeric or character type.

Scalar A single data object of any type. A scalar has a rank of zero.

Scalar variable A variable of scalar type.

Scope and scoping unit The part of a program in which a name has a defined meaning. The name may be a named constant, a variable, a function, a procedure, or dummy argument. The part of the program is one of a program unit or subprogram, a derived type definition or a procedure interface body. Scoping units cannot overlap, but one scoping unit may be contained in another. In the latter case we have an example of host association.

Shape The rank and extents of an array.

Shape conformance Generally means that two or more arrays have the same rank and extent.

Size The total number of elements in an array—the product of the extents.

Source file A file known to the operating system that contains the Fortran statements.

Statement An instruction in a programming language, normally classified as executable and nonexecutable.

Stride The increment in a subscript triplet.

Structure Either a scalar data object of derived type or a composite entity containing one or more subcomponents.

Structure component Component of a structure

Structure constructor Syntax that specifies a structure value or creates such a value

Subprogram A user written or supplied function or subroutine.

Subroutine A user subprogram that is invoked with the call statement. It can return one value, many values or no value at all to the calling program through the arguments.

Subscript A scalar integer expression used to select an element of an array

Subscript triplet A subscript triplet is a set of three values representing the lower bound of the array section, the upper bound of the array section, and the increment (stride) between them.

Substring A contiguous set of characters in a string.

Target A named data object associated with a pointer.

Transformational function An intrinsic function that is not elemental or inquiry.

Truncation For real numbers the approximation obtained by chopping off the fractional part of the number and working with the integer part.
For character variables removing one or more characters from a string.

Type-bound procedure Procedure that is bound to a derived type and referenced via an object of that type

Type compatible Compatibility of the type of one entity with respect to another for purposes such as argument association, pointer association, and allocation

Type declaration One of the nonexecutable statements in Fortran, and one of integer, real, complex, character, logical or type.

Underflow A condition where the result of an arithmetic expression is smaller than the minimum value in the range for that data type.

Use association Association between entities in a module and entities in a scoping unit or construct that references that module as specified by a USE statement

User defined type A data type that is defined by the user and not one of the intrinsic types.

Variable A data object that has an associated memory location whose value can be changed during program execution. A variable may be a scalar or an array.

Appendix B

ASCII Character Set

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	1
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	?	95	_	127	del

Appendix C

Intrinsic Functions and Procedures

This appendix has a brief coverage of some of the more commonly used intrinsic functions and procedures. Chap. 13 of the standard should be consulted for exhaustive coverage.

The following abbreviations and typographic conventions are used in this appendix.

Argument type and result type:

I	integer
R	real
C	complex
N	Numeric (any of integer, real, complex)
L	Logical
P	pointer
P*	polymorphic
T	target
DP	double precision
Char	character, length = 1.
S	character
Boz	boz-literal-constant
Co	coarray or coindexed object
Class	
A	indicates that the procedure is an atomic subroutine
E	indicates that the procedure is an elemental function
ES	indicates that the procedure is an elemental subroutine
I	indicates that the procedure is an inquiry function
PS	indicates that the procedure is a pure subroutine
S	indicates that the procedure is an impure subroutine
T	indicates that the procedure in a transformational function

See Chap. 12 for more information on these classifications.
Arguments in italics
ALL(Mask,*Dim*)
are optional arguments, i.e., Dim may be omitted in the example above.

Double precision

Before Fortran 90 if you required real variables to have greater precision than the default real then the only option available was to declare them as double precision. With the introduction of kind types with Fortran 90 the use of double precision declarations is not recommended, and instead real entities with a kind type offering more than the default precision should be used.

Kind optional argument

There are several functions that have an optional argument Kind, e.g., AINT(A,*Kind*). if Kind is absent the result is the same kind type as the first argument, in this case A. if Kind is present the result has the kind type specified by this argument.

Result type

When the result type is the same as the argument type then the result is not just the same type as the argument but also the same kind.

Miscellaneous rules

When the argument is Back it is of logical type.
When the argument is Count_Rate, Count_Max, Dim, Kind, Len, Order, N_Copies, Shape, Shift, Values it is of integer type.
When the argument is Mask it is of logical type.
When the argument is target it is of pointer or target type.
Fortran 2008 contained several changes to Fortran 2003 that affects intrinsic procedures.

The following functions can now have arguments of type complex, ACOS, ASIN, ATAN, COSH, SINH, TAN and TANH.
The intrinsic function ATAN2 can be referenced by the name ATAN.
The intrinsic functions LGE, LGT, LLE and LLT can have arguments of ASCII kind.
A BACK= argument has been added to the intrinsic functions MAXLOC and MINLOC.
A RADIX= argument has been added to the intrinsic function SELECTED_REAL_KIND.

ABS(A)

Yields the absolute value unless A is complex; see below.

Argument: A	Type: N
Result: As argument	Class: E

Note: if A is complex (x,y) then the functions returns $\sqrt{x^2 + y^2}$
Example: R1=ABS(A)
ACHAR(I)

Returns character in the ASCII character set.

Argument: I	Type: I
Result: Char	Class: E

Example: C=ACHAR(I)
ACOS(X)
Arccosine (inverse cosine).

Argument: X	Type: R
Result: As argument	Class: E

Note: $|x| \leq 1$
Example: Y=ACOS(X)
ACOSH(X)
Inverse hyperbolic cosine function.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Y = ACOSH(X)
ADJUSTL(String)
Adjust string left, removing leading blanks and inserting trailing blanks.

Argument: String	Type: S
Result: As argument	Class: E

Example: S=ADJUSTL(S)
ADJUSTR(String)
Adjust string right, removing trailing blanks and inserting leading blanks.

Argument: String	Type: S
Result: As argument	Class: E

Example: S=ADJUSTR(S)
AIMAG(Z)
Imaginary part of complex argument.

Argument: Z	Type: C
Result: As argument	Class: E

Example: Y=AIMAG(Z)
AINT(A,*Kind*)
Truncation.

Argument: A	Type: R
Result: As A	Class: E
Argument: Kind	Type: I

Example: Y=AINT(Z) and when Z=0.3 Y=0, when Z=2.73 Y=2.0, when Z=−2.73 Y=−2.0

ALL(Mask,*Dim*)

Determines whether all values are true in Mask along dimension Dim.

Argument: Mask	Type: L
Result: L	Class: T

Note: Dim must be a scalar in the range $1 \leq Dim \leq n$ where n is the rank of Mask. The result is scalar if Dim is absent or Mask has rank 1. Otherwise it works on the dimension Dim of Mask and the result is an array of rank n−1.

Example: T=ALL(M)

ALLOCATED(Array)

Returns true if array is allocated.

Argument: Array	Type: Any
Result: L	Class: I

Note: Array must be declared with the allocatable attribute.

Example: if (ALLOCATED(Array)) then …

ANINT(A,*Kind*)

Rounds reals, i.e., returns nearest whole number.

Argument: A	Type: R
Result: As A	Class: E

Example: Z=ANINT(A), if A = 5.63 Z = 6, if A = −5.7 Z = −6.0

ANY(Mask,*Dim*)

Determines whether any value is true in Mask along dimension Dim.

Argument: Mask	Type: L
Result: L	Class: T

Note: Mask must be an array. The result is a scalar if Dim is absent or if Mask is of rank 1. Otherwise it works on the dimension Dim of Mask and the result is an array of rank n−1.

Example: T=ANY(A)

ASIN(X)

Arcsine.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Z=ASIN(X)

ASINH(X)

Inverse hyperbolic sine function.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Y=ASINH(X)
ASSOCIATED(pointer, *target*)
Returns the association status of the pointer.

Argument: pointer	Type: P
Result: L	Class: I

Note:

1. if target is absent then the result is true if pointer is associated with a target, otherwise false.
2. if target is present and is a target, the result is true if pointer is currently associated with target and false if it is not.
3. if target is present and is a pointer, the result is true if both pointer and target are currently associated with the same target, and is false otherwise. if either pointer or target is disassociated the result is false.

Example: T=ASSOCIATED(P)
ATAN(X)
Arctangent.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Z=ATAN(X)
ATAN2(Y,X)
Arctangent of Y/X.

Arguments: Y	Type: R
Result: As arguments	Class: E

Example: Z=ATAN2(Y,X)
ATANH(X)
Inverse hyperbolic tangent function.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Y = ATANH(X)
ATOMIC_DEFINE (ATOM, VALUE)
Define a variable atomically.

Arguments: ATOM	Type: Co
VALUE	scalar and same type as ATOM
Result: N/A	Class: A

Note:

1. ATOM shall be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND, where ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are the named constants in the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (OUT) argument. If its kind is the same as that of VALUE or its type is logical, it becomes defined with the value of VALUE. Otherwise, it becomes defined with the value of INT (VALUE, ATOMIC_INT_KIND).
2. VALUE shall be scalar and of the same type as ATOM. It is an INTENT (IN) argument.

Example: CALL ATOMIC_DEFINE (I[2], 4) causes I on image 2 to become defined with the value 4.

ATOMIC_REF (VALUE, ATOM)
Reference a variable atomically.

Arguments: VALUE	Type: scalar and same type as ATOM
ATOM	Co
Result: N/A	Class: A

Note:

1. VALUE shall be scalar and of the same type as ATOM. It is an INTENT (OUT) argument. If its kind is the same as that of ATOM or its type is logical, it becomes defined with the value of ATOM. Otherwise, it is defined with the value of INT (ATOM, KIND (VALUE)).
2. ATOM shall be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND, where ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are the named constants in the intrinsic module ISO_FORTRAN_ENV. It is an INTENT (IN) argument.

BESSEL_J0(X)
Bessel function of the 1st kind, order 0.

Argument: X	Type: R
Result: As argument	Class: E

Example: Y = BESSEL_J0(1.0) has the value 0.765 (approximately)

BESSEL_J1(X)
Bessel function of the 1st kind, order 1.

Argument: X	Type: R
Result: As argument	Class: E

Example: Y = BESSEL_J1(1.0) has the value 0.440 (approximately).
BESSEL_JN (N, X)
Bessel functions of the 1st kind.

Arguments: N, X	Type: N is I, X is R
Result: as X	Class: E

Example: Y = BESSEL_JN (2, 1.0) has the value 0.115 (approximately).
BESSEL_JN (N1, N2, X) Bessel functions of the 1st kind.

Arguments: N1, N2, X	Type: N1, N2 are I, X is R
Result: is a real rank-one array with extent MAX(N2-N1+1, 0). Element I of the result value is a processor-dependent approximation to the Bessel function of the 1st kind and order N1+I−1 of X.	
Class: T	

BESSEL_Y0(X)
Bessel function of the 2nd kind, order 0.

Argument: X	Type: R
Result: As argument	Class: E

Example: Y = BESSEL_Y0(1.0) has the value 0.088 (approximately).
BESSEL_Y1(X)
Bessel function of the 2nd kind, order 1.

Argument: X	Type: R
Result: As argument	Class: E

Example: Y = BESSEL_Y 1(1.0) has the value −0.781 (approximately).
BESSEL_YN (N, X)
Bessel functions of the 2nd kind.

Arguments: N, X	Type: N is I, X is R
Result: Is same type and kind as X.	
Class: E	

Example: Y = BESSEL_YN (2, 1.0) has the value −1.651 (approximately).
BESSEL_YN (N1, N2, X)
Bessel functions of the 2nd kind.

Arguments: N1, N2, X	Type: N1, N2 are I, X is R
Result: is a real rank-one array with extent MAX(N2-N1+1, 0). Element I of the result value is a processor-dependent approximation to the Bessel function of the 2nd kind and order N1+I−1 of X.	
Class: T	

BGE(I, J)
Bitwise greater than or equal to.

Arguments: I,J	Type: I or Boz
Result: L	Class: E

The result is true if the sequence of bits represented by I is greater than or equal to the sequence of bits represented by J, according to the method of bit sequence comparison in 13.3.2 of Fortran 2008 standard; otherwise the result is false.

Example: If BIT SIZE (J) has the value 8, BGE (Z'FF', J) has the value true for any value of J. BGE (0,−1) has the value false.

BGT (I, J)
Bitwise greater than.

Arguments: I,J	Type: I or Boz
Result: L	Class: E

The result is true if the sequence of bits represented by I is greater than the sequence of bits represented by J, according to the method of bit sequence comparison in 13.3.2 of Fortran 2008 standard; otherwise the result is false.

Example: BGT (Z'FF', Z'FC') has the value true. BGT (0, -1) has the value false.

BLE (I, J)
Bitwise less than or equal to.

Arguments: I,J	Type: I or Boz
Result: L	Class: E

The result is true if the sequence of bits represented by I is less than or equal to the sequence of bits represented by J, according to the method of bit sequence comparison in 13.3.2 of Fortran 2008 standard; otherwise the result is false.

Example. BLE (0, J) has the value true for any value of J. BLE (−1, 0) has the value false.

BLT (I, J)
Bitwise less than.

Arguments: I,J	Type: I or Boz
Result: L	Class: E

The result is true if the sequence of bits represented by I is less than the sequence of bits represented by J, according to the method of bit sequence comparison in 13.3.2 of Fortran 2008 standard; otherwise the result is false.

Example: Example. BLT (0,−1) has the value true. BLT (Z'FF', Z'FC') has the value false.

BIT_SIZE(I)

Returns the number of bits, as defined by the numeric model for integer numbers in Chap. 5.

Argument: I	Type: I
Result: As argument	Class: I

Example: N_Bits=SIZE(I)

BTEST(I,Pos)

Returns true if the bit is set in the integer argument at the position given by the second argument.

Argument: I	Type: I
Result: L	Class: E

Example: T=BTEST(I,Pos)

CEILING(A,*Kind*)

Returns the smallest integer greater than or equal to the argument.

Argument: A	Type: R
Result: I	Class: E

Note:

if kind is present the result has the kind type parameter Kind.

Otherwise the result is of type default integer.

Example: I=CEILING(A) if A=12.21 then I=13, if A=−3.16 then I=−3

CHAR(I,*Kind*)

Returns the character in a given position in the processor collating sequence associated with the specified kind type parameter. Normally ASCII.

Argument: I	Type: I
Result: CHAR	Class: E

Example: C=CHAR(65) and for the ASCII character set C='A'.

CMPLX(X,Y,*Kind*)

Converts to complex from integer, real and complex.

Argument: X	Type: N
Result: C	Class: E

Note:

1. if X is complex and Y is absent it is as if Y were present with the value AIMAG(X).
2. if X is not complex and Y is absent, it is as if Y were present with the value 0.

Example: Z=CMPLX(X,Y)
COMMAND_ARGUMENT_COUNT ( )
Number of command arguments.

Arguments	None
Result: I	Class: T

The result value is equal to the number of command arguments available. If there are no command arguments available or if the processor does not support command arguments, then the result has the value zero. If the processor has a concept of a command name, the command name does not count as one of the command arguments.

Example: I=COMMAND_ARGUMENT_COUNT ( )
CONJG(Z)
Conjugate of a complex argument.

Argument: Z	Type: C
Result: As Z	Class: E

Example: Z1=CONJG(Z)
COS(X)
Cosine.

Argument: X	Type: R, C
Result: As argument	Class: E

Note: The arguments of all trigonometric functions should be in radians, not degrees.

Example: A=COS(X)
COSH(X)
Hyperbolic cosine.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Z=COSH(X)
COUNT(Mask,*Dim*)
Returns the number of true elements in Mask along dimension Dim.

Argument: Mask	Type: L
Result: I	Class: T

Note: Dim must be a scalar in the range $1 \leq Dim \leq n$, where n is the rank of Mask. The result is scalar if Dim is absent or Mask has rank 1. Otherwise it works on the dimension Dim of Mask and the result is an array of rank n−1.

Example: N=COUNT(A)

CPU_TIME(Time)

Returns the processor time.

Argument: Time	Type: R
Result: N/A	Class: S

Example: call CPU_TIME(Time)

CSHIFT(Array,Shift,*Dim*)

Circular shift on a rank 1 array or rank 1 sections of higher-rank arrays.

Argument: Array	Type: Any
Result: As Array	Class: T

Note: Array must be an array, Shift must be a scalar if Array has rank 1, otherwise it is an array of rank n−1, where n is the rank of Array. Dim must be a scalar with a value in the range $1 \leq Dim \leq n$.

Example: Array=CSHIFT(Array,10)

DATE_AND_TIME(*Date,Time,Zone,Values*)

Returns the current date and time (compatible with ISO 8601:1988).

Argument: Date	Type: S
Result: N/A	Class: S

Time and Zone are of type S.

Note:

1. Date is optional and must be scalar and 8 characters long in order to return the complete value of the form CCYYMMDD, where CC is the century, YY is the year, MM is the month and DD is the day. It is intent(out).
2. Time is optional and must be scalar and 10 characters long in order to return the complete value of the form hhmmss.sss where hh is the hour, mm is the minutes and ss.sss is the seconds and milliseconds. It is intent(out).
3. Zone is optional and must be scalar and must be 5 characters long in order to return the complete value of the form hhmm where hh and mm are the time differences with respect to Coordinated Universal Time in hours and minutes. It is intent(out).
4. Values is optional and a rank 1 array of size 8. It is intent(out). The values re turned are as follows:
   Values(1) = the year
   Values(2) = the month
   Values(3) = the day

Values(4) = the time with respect to Coordinated Universal Time in minutes.
Values(5) = the hour (24 hour clock)
Values(6) = the minutes
Values(7) = the seconds
Values(8) = the milliseconds in the range 0–999.

Example: call DATE_TIME(D,T,Z,V)
DBLE(A)
Converts to double precision from integer, real, and complex

Argument: A	Type: N
Result: DP	Class: E

Example: D=DBLE(A)
DIGITS(X)
Returns the number of significant digits of the argument as defined in the numeric models for integer and reals in Chap. 5.

Argument: X	Type: I,R
Result: I	Class: I

Example: I=DIGITS(X)
DIM(X,Y)
Returns first argument minus minimum of the two arguments: X -MIN(X,Y).

Argument: X	Type: I
Result: As arguments	Class: E

Example: Z=DIM(X,Y)
DOT_PRODUCT(Vector_1,Vector_2)
Performs the mathematical dot product of two rank 1 arrays.

Argument: Vector_1	Type: N
Result: As arguments	Class: T

Vector_2 is as Vector_1.
Note:

1. if Vector_1 is of type integer or real the result has the value SUM(Vector_1*Vector_2).
2. if Vector_1 is complex the result has the value SUM(CONJG(Vector_1)*Vector_2).
3. if Vector_1 is logical the result has the value ANY(Vector_1 .AND. Vector_2).

Example: A=DOT_PRODUCT(X,Y)
DPROD(X,Y)
double precision product of two reals.

Argument: X	Type: R
Result: DP	Class: E

Example: D=DPROD(X,Y)

DSHIFTL (I, J, SHIFT)
Combined left shift.

Arguments: I,J	Type: I or Boz
SHIFT	Type: I

Result: Same as I if I is of type integer; otherwise, same as J. If either I or J is a boz-literal-constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The rightmost SHIFT bits of the result value are the same as the leftmost bits of J, and the remaining bits of the result value are the same as the rightmost bits of I. This is equal to IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT SIZE (J)-SHIFT)). The model for the interpretation of an integer value as a sequence of bits is in section 13.3 of Fortran 2008 standard.

Class: E

Examples: DSHIFTL (1, 2**30, 2) has the value 5 if default integer has 32 bits.
DSHIFTL (I, I, SHIFT) has the same result value as ISHFTC (I, SHIFT).
DSHIFTR (I, J, SHIFT)
Combined right shift.

Arguments: I,J	Type: I or Boz
SHIFT	Type: I

Result: Same as I if I is of type integer; otherwise, same as J. If either I or J is a boz-literal-constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The leftmost SHIFT bits of the result value are the same as the rightmost bits of I, and the remaining bits of the result value are the same as the leftmost bits of J. This is equal to IOR (SHIFTL (I, BIT SIZE (I)-SHIFT), SHIFTR (J, SHIFT)). The model for the interpretation of an integer value as a sequence of bits is in 13.3 of Fortran 2008 standard.

Class: E

Examples. DSHIFTR (1, 16, 3) has the value 229 +2 if default integer has 32 bits.
DSHIFTR (I, I, SHIFT) has the same result value as ISHFTC (I,-SHIFT).
EOSHIFT(Array, Shift, *Boundary, Dim*)
End of shift of a rank 1 array or rank 1 section of a higher-rank array.

Argument: Array	Type: Any
Result: As Array	Class: T

Boundary is as Array.

Note: Array must be an array, Shift must be a scalar if Array has rank 1, otherwise it is an array of rank n−1, where n is the rank of Array. Boundary must be scalar if Array has rank 1, otherwise it must be either scalar or of rank −1. Dim must be a scalar with a value in the range $1 \leq Dim \leq n$.

Example: A=EOSHIFT(A,Shift)

EPSILON(X)

Smallest difference between two reals of that kind. See Chap. 5 and real numeric model.

Argument: X	Type: R
Result: As argument	Class: I

Example: Tiny=EPSILON(X)

ERF (X)

Error function.

Argument: X	Type: R
Result: As X	Class: E

Example: Y = ERF(1.0) has the value 0.843 (approximately).

ERFC (X)

Complementary error function.

Argument: X	Type: R
Result: As X	Class: E

Example: Y = ERFC (1.0) has the value 0.157 (approximately).

ERFC_SCALED (X)

Scaled complementary error function.

Argument: X	Type: R
Result: As X	Class: E

Example: Y = ERFC_SCALED (20.0) has the value 0.0282 (approximately).

EXECUTE_COMMAND_LINE(COMMAND,*WAIT,EXITSTAT,CMDSTAT. CMDMSG*)

Execute a command line.

Argument: COMMAND – shall be a default character scalar. It is an INTENT (IN) argument. Its value is the command line to be executed. The interpretation is processor dependent.

Argument: WAIT – (optional) shall be a default logical scalar. It is an INTENT (IN) argument. If WAIT is present with the value false, and the processor supports asynchronous execution of the command, the command is executed asynchronously; otherwise it is executed synchronously.

Argument: EXITSTAT – (optional) shall be a default integer scalar. It is an INTENT (INOUT) argument. If the command is executed synchronously, it is assigned the value of the processor-dependent exit status. Otherwise, the value of EXITSTAT is unchanged.
Argument: CMDSTAT – (optional) shall be a default integer scalar. It is an INTENT (OUT) argument. It is assigned the value −1 if the processor does not support command line execution, a processor-dependent positive value if an error condition occurs, or the value −2 if no error condition occurs but WAIT is present with the value false and the processor does not support asynchronous execution. Otherwise it is assigned the value 0.
Argument: CMDMSG – (optional) shall be a default character scalar. It is an INTENT (INOUT) argument. If an error condition occurs, it is assigned a processor-dependent explanatory message. Otherwise, it is unchanged.

Class: S

Example: CALL EXECUTE_COMMAND_LINE('pwd') will print the full pathname of the current directory under Unix and an error message from Windows.

EXP(X)

Exponential, e^x.

Argument: X	Type: R, C
Result: As argument	Class: E

Example: Y=EXP(X)

EXPONENT(X)

Returns the exponent component of the argument. See Chap. 5 and the real numeric model.

Argument: X	Type: R
Result: I	Class: E

Example: I=EXPONENT(X)

EXTENDS_TYPE_OF (A, MOLD)

Query dynamic type for extension.

Arguments: A, Mold	Type: P*
Result: L	Class: I

If MOLD is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable variable, the result is true; otherwise if A is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable variable, the result is false; otherwise if the dynamic type of A or MOLD is extensible, the result is true if and only if the dynamic type of A is an extension type of the dynamic type of MOLD; otherwise the result is processor dependent.

Example:

```
if (extends_type_of(a, mold) then
print*,'dynamic type of a is an extension of dynamic type of mold'
endif
```

FINDLOC (ARRAY, VALUE, *DIM, MASK, KIND, BACK*)
FINDLOC (ARRAY, VALUE, *MASK, KIND, BACK*)

Location(s) of a specified value.

Argument: ARRAY	Type: shall be an array of intrinsic type
Argument: VALUE	Type: shall be scalar and in type conformance with ARRAY, as specified in Table 7.2 for relational intrinsic operations 7.1.5.5.2).
Argument: DIM	Type: shall be an integer scalar with a value in the range 1 DIM `n`, where `n` is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.
Argument: MASK	Type: (optional) shall be of type logical and shall be conformable with ARRAY.
Argument: KIND	Type: (optional) shall be a scalar integer constant expression.
Argument: BACK	Type:(optional) shall be a logical scalar.
	Class: T

FLOOR(A, *Kind*).
Returns the greatest integer less than or equal to the argument

Argument: A	Type: R
Result: I	Class: E

Note:

If kind is present the result has the kind type parameter Kind, otherwise the result is of type default integer.

Example: I=FLOOR(A) and when A=5.2 I has the value 5, when A=−9.7 I has the value −10

FRACTION(X)

Returns the fractional part of the real numeric model of the argument See Chap. 5 and the real numeric model.

Argument: X	Type: R
Result: As X	Class: E

Example: F=FRACTION(X)
GAMMA (X)
Gamma function.

Argument: X	Type: R
Result: As X	Class: E

Example: Y = GAMMA(1.0) has the value 1.000 (approximately).

GET_COMMAND (*COMMAND, LENGTH, STATUS*)

Query program invocation command.

GET_COMMAND_ARGUMENT (NUMBER, *VALUE, LENGTH, STATUS*)

Query arguments from program invocation.

GET_ENVIRONMENT_VARIABLE (NAME, *VALUE, LENGTH, STATUS, TRIM_NAME*)

Query environment variable.

HUGE(X)

Returns the largest number for the kind type of the argument. See Chap. 5 and the real and integer numeric models.

Argument: X	Type: I,R
Result: As argument	Class: I

Example: H=HUGE(X)

HYPOT (X, Y)

Euclidean distance function.

Arguments: X,Y	Type: R
Result: R	Class: E

Example: H = HYPOT(3.0, 4.0) has the value 5.0 (approximately).

IACHAR(C)

Returns the position of the character argument in the ASCII collating sequence.

Argument: C	Type: Char
Result: I	Class: E

Example: I=IACHAR('A') returns the value 65.

IALL (ARRAY, *DIM, MASK*) or IALL (ARRAY, *MASK*)

Reduce array with bitwise AND operation.

IAND(I,J)

Performs a logical AND on the arguments.

Argument: I	Type: I
Result: As arguments	Class: E

Example: K=IAND(I,J)

IANY (ARRAY, *DIM, MASK*) or IANY (ARRAY, *MASK*)

Reduce array with bitwise OR operation.

IBCLR(I,Pos)
Clears one bit of the argument to zero.

Argument: I	Type: I
Result: As I	Class: E

Note: $0 \le Pos < BIT\_SIZE(I)$
Example: I=IBCLR(I,Pos)
IBITS(I,Pos,Len)
Returns a sequence of bits.

Argument: I	Type: I
Result: As I	Class: E

Note: $0 \le Pos$ and $(Pos + Len) \le BIT\_SIZE(I)$ and $Len \ge 0$.
Example: Slice=IBITS(I,Pos,Len)
IBSET(I,Pos)
Sets one bit of the argument to one.

Argument: I	Type: I
Result: As I	Class: E

Note: $0 \le Pos < BIT\_SIZE(I)$.
Example: I=IBSET(I,Pos)
ICHAR(C)
Returns the position of a character in the processor collating sequence associated with the kind type parameter of the argument. Normally the position in the ASCII collating sequence.

Argument: C	Type: CHAR
Result: I	Class: E

Example: I=ICHAR('A') would return the value 65 for the ASCII character set.
IEOR(I,J)
Performs an exclusive OR on the arguments.

Argument: I	Type: I
Result: As I	Class: E

Example: I=IEOR(I,J)
IMAGE_INDEX (COARRAY, SUB)
Convert cosubscripts to image index.

Argument: COARRAY	Type: Co
Argument: SUB	Rank-one integer array
Result: I	Class: I

Example:

```
integer, codimension[0:*]:: x
integer, dimension(10,15), codimension[3,0:1,−1:*]:: z
   print*, image_index(x,(/0/)); print*, image_index(z,(/2,0,−1/))
```

would print 1 and 2 respectively.

INDEX(String,Substring,*Back*)

Locates one substring in another, i.e., returns position of Substring in character expression String.

Argument: String	Type: S
Result: I	Class: E

Substring is of type S.

Note:

1. if Back is absent or present with the value .FALSE. then the function returns the start position of the first occurrence of the substring. if LEN(Substring) = 0 then one is returned.
2. if Back is present with the value .TRUE. then the function returns the start position of the last occurrence of the substring. if LEN(Substring) = 0 then the value (LEN(String) + 1) is returned.
3. if the substring is not found the result is zero.
4. if LEN(String) < LEN(Substring) the result is zero.

Example:

```
where=INDEX('Hello world Hello','Hello')
```

The result 2 is returned.

```
where=INDEX('Hello world Hello','Hello',.TRUE.)
```

The result 14 is returned.

INT(A,*Kind*)

Converts to integer from integer, real, and complex.

Argument: A	Type: N
Result: I	Class: E

Example: I=INT(F)

IOR(I,J)

Performs an inclusive OR on the arguments.

Argument: I	Type: I
Result: As I	Class: E

Example: I=IOR(I,J)

IPARITY (ARRAY, DIM, *MASK*) or IPARITY (ARRAY, *MASK*)

Reduce array with bitwise exclusive OR operation.

ISHFT(I, Shift)

Performs a logical shift. The bits of I are shifted by Shift positions.

Argument: I	Type: I
Result: As I	Class: E

Note: $|Shift| \leq BIT\_SIZE(I)$

Example: I=ISHIFT(I,Shift).

ISHFTC(I,Shift,*Size*)

Performs a circular shift of the rightmost bits. The Size rightmost bits of I are circularly shifted by Shift positions.

Argument: I	type: I
Result: I	Class: E

Note:

$|Shift| < Size$

$0 \leq Size \leq BIT\_SIZE(I)$.

if Size is absent it is as if it were present with the value of $BIT\_SIZE(I)$.

if Shift is positive the shift is to the left.

if Shift is negative the shift is to the right.

if Shift is zero no shift is performed.

Example: I=ISHFTC(I,Shift,Size)

IS_CONTIGUOUS (ARRAY)

Test contiguity of an array.

Argument: ARRAY	Type: any
Result: L	Class: I

Example:

```
integer,target, dimension(10)::a
integer,pointer,dimension(:) :: p
      p=> a(1:10:2); print*,is_contiguous(p)
```

would print 'F'

IS_IOSTAT_END (I)

Test IOSTAT value for end-of-file.

Argument: I	Type: I
Result: L	Class: E

Example:

IS_IOSTAT_END(I) returns value true if I is an I/O status value that corresponds to an end-of-file condition, and false otherwise.

```
read(unit= 1, fmt=*, iostat=ist)y(I)
…
if(is_iostat_end(ist)) then
      print*,'end of file!'
endif
```

IS_IOSTAT_EOR (I)
Test IOSTAT value for end-of-record.

Argument: I	Type: I
Result: L	Class: E

Example: IS_IOSTAT_EOR(I) returns value true if I is an I/O status value that corresponds to an end-of-record condition, and false otherwise.

KIND(X)
Returns the KIND type parameter of the argument.

Argument: X	Type: Any
Result: I	Class: I

Example: I=KIND(X)

LBOUND(Array,*Dim*)
Returns the lower bounds for each dimension of the array argument or a specified lower bound.

Argument: Array	Type: Any
Result: I	Class: I

Note:

$1 \leq Dim \leq n$, where n is the rank of Array. The result is scalar if Dim is present otherwise the result is an array of rank 1 and size n.

The result is scalar if Dim is present, otherwise a rank 1 array and size n.

Example: I=LBOUND(Array)

LCOBOUND (COARRAY, *DIM, KIND*)
Lower cobound(s) of a coarray.

Argument: COARRAY	Type: co
Argument: DIM (optional)	Type: I
Argument: KIND (optional)	Type: I
Result: I	Class: I

Example:

```
INTEGER, CODIMENSION[:,:], ALLOCATABLE::A
  ALLOCATE(A[2:3,7:*])
  LCBOUND (A) is [2,7] and LCOBOUND(A,DIM=2) is 7
```

LEADZ (I)

Number of leading zero bits.

Argument: I	Type: I
Result: I	Class: E

Example: LEADZ (1) has the value 31 if BIT SIZE (1) has the value 32.
LEN(String)
Length of a character entity.

Argument: String	Type: S
Result: I	Class: I

Example: I=LEN(String)
LEN_TRIM(String)
Length of character argument less the number of trailing blanks.

Argument: String	Type: S
Result: I	Class: E

Example: I=LEN_TRIM(String)
LGE(String_1,String_2)
Lexically greater than or equal to and this is based on the ASCII collating sequence.

Argument: String_1	Type: S
Result: L	Class: E

String_2 is of type S.
Example: L=LGE(S1,S2)
LGT(String_1 ,String_2)
Lexically greater than and this is based on the ASCII collating sequence.

Argument: String_1	Type: S
Result: L	Class: E

Example: L=LGT(S1,S2)
LLE(String_1, String_2)
Lexically less than or equal to and this is based on the ASCII collating sequence.

Argument: String_1	Type: S
Result: L	Class: E

String_2 is of type S.
Example: L=LLE(S1,S2)
LLT(String_ 1, String_2)

Lexically less than and this is based on the ASCII collating sequence.

Argument: String_1	Type: S
Result: L	Class: E

Example: L=LLT(S1,S2)
LOG(X)
Natural logarithm, loge x.

Argument: X	Type: R, C
Result: As argument	Class: E

Example: Y=LOG(X)
LOG_GAMMA (X)
Logarithm of the absolute value of the gamma function.

Argument: X	Type: R
Result: R	Class: E

Example: LOG_GAMMA (3.0) has the value 0.693 (approximately)
LOG10(X)
common logarithm, log 10.

Argument: X	Type: R
Result: As argument	Class: E

Example: Y=LOG10(X)
LOGICAL(L,Kind)
Converts between different logical kind types, i.e., performs a type cast.

Argument: L	Type: L
Result: L	Class: E

Example: L=LOGICAL(K,*Kind*)
MASKL (I, *KIND*)
Left justified mask.

Argument: I	Type: I
Result: I	Class: E

Example: MASKL (4) has the value SHIFTL (15, BIT_SIZE (0) - 4)
MASKR (I, *KIND*)
Right justified mask.

Argument: I	Type: I
Result: I	Class: E

Example: MASKR(4) has the value 15.
MATMUL(Matrix_1 ,Matrix_2)
Performs mathematical matrix multiplication of the array arguments.

Argument: Matrix_1	Type: N,L
Result: As arguments	Class: T

Matrix_2 is as Matrix_1.
Note:

1. Matrix_1 and Matrix_2 must be arrays of rank 1 or 2. if Matrix_1 is of numeric type so must Matrix_2.
2. if Matrix_1 has rank 1, Matrix_2 must have rank 2.
3. if Matrix_2 has rank 1, Matrix_1 must have rank 2.
4. The size of the first dimension of Matrix_2 must equal the size of the last dimension of Matrix_1.
5. if Matrix_1 has shape (n,m) and Matrix_2 has shape (m,k) the result has shape (n,k).
6. if Matrix_1 has shape (m) and Matrix_2 has shape (m,k) the result has shape (k).
7. if Matrix_1 has shape (n,m) and Matrix_2 has shape (m) the result has shape (n).

Example: R=MATMUL(M_1,M_2)
MAX(A1,A2,A3,…)
Returns the largest value.

Argument: A1	Type: I,R,S
Result: As arguments	Class: E

A2, A3,.. are as A1.
Example: A=MAX(A1,A2,A3,A4)
MAXEPONENT(X)
Returns the maximum exponent. See Chap. 5 and numeric models.

Argument: X	Type: R
Result: I	Class: I

Example: I=MAXEXPONENT(X)
MAXLOC(ARRAY,*Dim, Mask, Kind, Back*)
Determine the location of the first element of Array having the maximum value of the elements identified by Mask if present.

Argument: Array	Type: I,R
Result: I	Class: T

Note:

0. Normally in Fortran if you omit an optional argument you must use keywords for the rest. This intrinsic breaks this rule and DIM can be omitted and it is not necessary to use a keyword with Mask.
1. Array must be an array.
2. Mask must be conformable with Array
3. The result is an array of rank 1 and of size equal to the rank of Array.
4. if Dim is present the result is an array of the rank of Array reduced by one and with the shape of Array without the dimension Dim.

Example:
A=(/5,6,7,8/)
I=MAXLOC(A)
is (4), which is the subscript of the location of the first occurrence of the maximum value in the rank 1 array.

if $A = \begin{pmatrix} 1 & 8 & 5 \\ 9 & 3 & 6 \\ 4 & 2 & 7 \end{pmatrix}$

I = MAXLOC(A,dim=1)
is (2,1,3) returning the position of the largest in each column.
I = MAXLOC(A,dim=2)
is (2,1,3) returning the position of the largest in each row.
MAXVAL(Array,*Dim,Mask*)
Returns the maximum value of the elements of Array along dimension Dim corresponding to the true elements of Mask.

Argument: Array	Type: I,R,S
Result: As argument	Class: T

Note:

$1 \le Dim \le n$, where n is the rank of Array. The result is scalar if Dim is absent, or Array has rank 1. Otherwise the result is an array of rank −1.

if Array has size zero then the result is the largest negative number supported by the processor for the corresponding type and kind of Array.

Example:
MAXVAL((/1,2,3/)) returns the value 3.
MAXVAL(C,MASK=C < 0.0) returns the maximum of the negative elements of C.

For $B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

MAXVAL(B,DIM=1) returns (2,4,6)
MAXVAL(B,DIM=2) returns (5,6)
MERGE(True,False,Mask)
Chooses alternative values according to the value of a mask.

Argument: True	Type: Any
Result: As True	Class: E

Example: for

For $True = \begin{pmatrix} 2 & 6 & 10 \\ 4 & 8 & 12 \end{pmatrix}$, $False = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$ and $Mask = \begin{pmatrix} T & F & T \\ F & T & F \end{pmatrix}$

The result is $\begin{pmatrix} 2 & 5 & 10 \\ 3 & 8 & 11 \end{pmatrix}$

MERGE_BITS (I, J, MASK)
Merge of bits under mask.

Argument: I	Type: I or Boz
Argument: J	I or Boz
Argument: MASK	I or Boz
Result: same as I if Integer, otherwise same as J.	
	Class: E

Example: MERGE_BITS(14,18,22) has the value 6.
MIN(A1,A2,A3,…)
Chooses the smallest value.

Argument: A1	Type: I,R,S
Result: As arguments	Class: E

ww

Example: Y=MIN(X1,*X*2,X3,X4,X5)
MINEXPONENT(X)
Returns the minimum exponent. See Chap. 5 and numeric models.

Argument: X	Type: R
Result: I	Class: I

Example: I=MINEXPONENT(X)
MINLOC(Array,*Dim, Mask, Kind, Back*)
Determine the location of the first element of Array having the minimum value of the elements identified by Mask.

Argument: Array	Type: I,R
Result: I	Class: T

Note:

0. Normally in Fortran if you omit an optional argument you must use keywords for the rest. This intrinsic breaks this rule and Dim can be omitted and it is not necessary to use a keyword with Mask.
1. Array must be an array.
2. Mask much be conformable with Array.
3. The result is an array of rank 1 and of size equal to the rank of Array.
4. if DIM is present the result is an array of the rank of Array reduced by one and with the shape of Array without the dimension DIM.

Example: I=MINLOC(Array)

In the above example if Array is a rank 2 array of shape (5,10) and the smallest value is in position (2,1) then the result is the rank 1 array I with shape (2) and I(1)=2 and I(2)=1.

See MAXLOC for further examples.

MINVAL(Array,*Dim,Mask*)

Returns the minimum value of the elements of Array along dimension Dim corresponding to the true elements of Mask.

Argument: Array	Type: I,R,S
Result: As Array	Class: T

Note:1 ≤ *Dim* ≤ *n*, where n is the rank of Array. The result is scalar if Dim is absent, or Array has rank 1. Otherwise the result is an array of rank n–1.

if Array has size zero then the result is the largest negative number supported by the processor for the corresponding type and kind of Array.

Example:

MINAL((/1,2,3/)) returns the value 1.

MINVAL(C,MASK=C > 0.0) returns the minimum of the positive elements of C.

For $B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

MINVAL(B,DIM=1) returns (1,3,5).

MINVAL(B,DIM=2) returns (1,2).

MOD(A,B)

Returns the remainder when first argument divided by second.

Argument: A	Type: I, R
Result: As arguments	Class: E

Note: if B=0 the result is processor dependent. For B ≠ 0 the result is A - INT (A/B) * B.

Example: R=MOD(A,B)
if A=8 and B=5 then R=3
if A=−8 and B=5 then R=−3
if A=8 and B=−5 then R=3
if A=−8 and B=−5 then R=−3
MODULO(A,B)
Returns the modulo of the arguments.

Argument: A	Type: I,R
Result: As A	Class: E

Note:

1. if B=0 then the result is processor dependent.
2. integer A
   The result is R where A= Q * B + R and Q is integer
   for B>0, $0 \le R < B$
   for B < 0, $B < R \le 0$
3. real A

   The result is A - FLOOR(A/B) * B.
   Example: R=MODULO(A,B)
   if A=8 and B=5 then R=3
   if A=−8 and B=5 then R=2
   if A=8 and B=−5 then R=−2
   if A=−8 and B=−5 then R=−3
   MOVE_ALLOC (FROM, TO)
   Move an allocation.

Argument: FROM	May be any type and rank. It shall be allocatable. It is INTENT(INOUT).
Argument: TO	Type compatible with FROM and same rank. It shall be allocatable.
	Class: Pure subroutine

Example:

```
integer, dimension(:), allocatable:: b,c
   allocate(b(1:12))
   b(2) = 24
   call mov_alloc(from=b, to=c)
! b is unallocated
! c is allocated with bounds (1:12) and c(2) == 24
```

MVBITS(From,F_Pos,Len,To,T_Pos)
Copies a sequence of bits from one data object to another.

Argument: From	Type: I
Result: N/A	Class: S

All arguments are of integer type.

Note:

From must be intent(in).
F_Pos must be intent(in), F_Pos ≥ 0, F_Pos+Len ≤ BIT_SIZE(From).
Len must be intent(in), Len ≥ 0.
To must be intent(inout).
T_Pos must be intent(in), T_Pos ≥ 0, T_Pos + Len ≤ BIT_SIZE(To).

Example: call MVBITS(F,FP,L,T,TP)
NEAREST(X,Next)
Returns the nearest different number. See Chap. 5 and the real numeric model.

Argument: X	Type: R
Result: As X	Class: E

Next is of type R.
Example: N=NEAREST(X,Next)
NEW_LINE (A)
Returns newline character used for formatted stream output.

Argument: A	Type: Char
Result: Char	Class: I

Example:

```
open(2,file='nline.txt', access='stream', form='formatted')
write(2,'(a)')'hola'//new_line('a')//'mundo'
```

will write 2 lines to the file nline.txt.

NINT(A,*Kind*)
Yields nearest integer.

Argument: A	Type: RI
Result: I	Class: E

Note:

1. A > 0, the result is INT(A+0.5).
2. A ≤ 0, the result is INT(A−0.5).

Example: I=NINT(X)
NORM2 (X, *DIM*)
L2 norm of an array.

Argument: X	Type: R array
Argument: DIM	Type: DIM (optional) shall be an integer scalar with a value in the range 1 <= DIM <= n, where n is the rank of X. The corresponding actual argument shall not be an optional dummy argument.
Result: R	Class: T

Note:

Case (i): The result of NORM2 (X) has a value equal to a processor-dependent approximation to the generalized L2 norm of X, which is the square root of the sum of the squares of the elements of X.
Case (ii): If DIM is present the array is reduced as for SUM(X,DIM) except that NORM2 is applied to the reduced vectors.

Examples:
NORM2([3.0, 4.0]) is 5.0.
If X has the value 1.0 2.0 3.0 4.0
then NORM2(X,DIM=1) is [3.162, 4.472] and NORM2(X,DIM=2) is [2.236,5.0] approximately.

NOT(I)
Returns the logical complement of the argument.

Argument: I	Type: I
Result: As I	Class: E

Example: I=NOT(I)
NULL(Mold)
Returns a disassociated pointer.

Argument: Mold	Type: P
Result: As argument	Class: T

Note:

if the argument Mold is present the result is the same as Mold.
Otherwise it is determined by context.

Example: real, pointer :: P=>NULL()
NUM_IMAGES ( )
Number of images.
Argument: None
Result: I
Class: T

Example:

```
PRINT*, 'number of images =',NUM_IMAGES( )
```

PACK(Array,Mask, *Vector*)
Packs an array into an array of rank 1, under the control of a mask.

Argument: Array	Type: Any
Result: As Array	Class: T

Note:

1. Array must be an array.
2. Mask be conformable with Array.
3. Vector must have rank 1 and have at least as many elements as there are TRUE elements in Mask.
4. if Mask is scalar with the value TRUE. Vector must have at least as many elements as there are in Array.
5. The result is an array of rank 1.
6. if Vector is present the result size is that of Vector.
7. if Vector is not present the result size is t, the number of TRUE elements in Mask, unless Mask is scalar with a value TRUE in which case the result size is the size of Array.

Example: R=PACK(A,M)
PARITY (MASK, *DIM*)
Reduce array with .NEQV. operation.

Argument: MASK	Type: L array
Argument: DIM	I scalar in the range 1 <= DIM <=n where n is rank of MASK.

Example:
If T has the value true and F has the value false
PARITY([T,T,T,F]) is true.
POPCNT (I)
Number of one bits in the sequence of bits of I.

Argument: I	Type: I
Result: I	Class: E

Example:
POPCNT ([1, 2, 3, 4, 5, 6, 7]) has the value [1, 1, 2, 1, 2, 2, 3].
POPPAR (I)
Returns the parity of the bit count of an integer expressed as 0 or 1. POPPAR (I) has the value 1 if POPCNT (I) is odd, and 0 if POPCNT (I) is even.

Argument: I	Type: I
Result: I	Type: E

Example:
POPPAR ([1, 2, 3, 4, 5, 6, 7]) has the value [1, 1, 0, 1, 0, 0, 1].
PRECISION(X)

Returns the decimal precision of the argument. See Chap. 5 and numeric models.

Argument: X	Type: R, C
Result: I	Class: I

Example: I=PRECISION(X)

PRESENT(A)

Returns whether an optional argument is present.

Argument: A	Type: Any
Result: L	Class: I

Note: A must be an optional argument of the procedure in which the PRESENT function reference appears.

Example: if (PRESENT(X)) then …

PRODUCT(Array,*Dim,Mask*)

The product of all of the elements of Array along the dimension Dim corresponding to the TRUE elements of Mask.

Argument: Array	Type: N
Result: As Array	Class: T

Note:

1. Array must be an array.
2. $1 \leq \text{Dim} \leq n$ where n is the rank of Array.
3. Mask must be conformable with Array.
4. result is scalar if Dim is absent, or Array has rank 1, otherwise the result is an array of rank n–1.

Example:

1. PRODUCT((/1,2,3/)) the result is 6.
2. PRODUCT(C,Mask=C > 0.0) forms the product of the positive elements of C.
3. if $B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

PRODUCT(B,DIM=1) is (2,12,30) and

PRODUCT(B,DIM=2) is (15,48)

RADIX(X)

Returns the base of the numeric argument. See Chap. 5 and numeric models.

Argument: X	Type: I,R
Result: I	Class: I

Example: Base=RADIX(X)

RANDOM_NUMBER(X)

Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \le x < 1$

Argument: X	Type: R
Result: N/A	Class: S

Note: X is intent(out).

Example: call RANDOM_NUMBER(X)

RANDOM_SEED(*Size,Put,Get*)

Restarts (seeds) or queries the pseudorandom generator used by RANDOM_NUMBER.

Argument: Size	Type: I
Result: N/A	Class: S

All arguments are of integer type.

Note:

1. Size is intent(out). It is set to the number N of integers that the processor uses to hold the value of the seed.
2. Put is intent(in). It is an array of rank 1 and size ≥ N. It is used by the processor to set the seed value.
3. Get is intent(out). It is an array of rank 1 and size ≥ N. It is set by the processor to the current value of the seed.

Example: call RANDOM_SEED

RANGE(X)

Returns the decimal exponent range of the real argument. See Chap. 5 and the numeric model representing the argument.

Argument: X	Type: N
Result: I	Class: I

Example: I=RANGE(N)

REAL(A,*Kind*)

Converts to real from integer, real or complex.

Argument: A	Type: N
Result: R	Class: E

Example: X=real(A)

REPEAT(String,N_Copies)

Concatenates several copies of a string.

Argument: String	Type: S
Result: S	Class: T

Example: New_S=REPEAT(S,10)

RESHAPE(Source,Shape,*Pad,Order*)

Constructs an array of a specified shape from the elements of a given array.

Argument: Source	Type: Any
Result: As Source	Class: T

Note:

1. Source must be an array. if Pad is absent or of size zero the size of Source must be ≥ PRODUCT(Shape).
2. Shape must be a rank 1 array and $0 \leq \text{size} < 8$.
3. Pad must be an array.
4. Order must have the same shape as Shape and its value must be a permutation of (1,2,… ,n) where n is the size of Shape. if absent it is as if it were present with the value (1,2,…,n).
5. The result is an array of shape, Shape.

Example:

RESHAPE((/1,2,3,4,5,6/),(/2,3/)) has the value $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

RESHAPE((/1,2,3,4,5,6/) , (/2,4/) , (/0,0/) , (/2,1/)) has the value $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{pmatrix}$

RRSPACING(X)

Returns the reciprocal of the relative spacing of model numbers near the argument value. See Chap. 5 and the real numeric model.

Argument: X	Type: R
Result: As X	Class: E

Example: Z=RRSPACING(X)

SAME_TYPE_AS (A, B)

Query dynamic types for equality. If the dynamic type of A or B is extensible, the result is true if and only if the dynamic type of A is the same as the dynamic type of B. If neither A nor B has extensible dynamic type, the result is processor dependent.

Argument: A	An object of extensible declared type or unlimited polymorphic. If it is a pointer, it shall not have an undefined association status.
Argument: B	An object of extensible declared type or unlimited polymorphic. If it is a pointer, it shall not have an undefined association status.
Result: L	Type: I

SCALE(X,I)

Returns X * b^I where b is the base in the model representation of X. See Chap. 5 and the real numeric model.

Argument: X	Type: R
Result: As X	Class: E

I is of integer type.

Example: Z=SCALE(X,I)

SCAN(String,Set,*Back*)

Scans a string for any one of the characters in a set of characters.

Argument: String	Type: S
Result: I	Class: E

Note:

1. The default is to scan from the left, and will only be from the right when Back is present and has the value TRUE.
2. Zero is returned if the scan fails.

Example: W=SCAN(String,Set)

SELECTED_CHAR_KIND (NAME)

Returns the kind value for the character set whose name is given by the character string NAME or −1 if not supported.

Argument: NAME	Type: Char
Result: I	Class: T

Note:

If NAME has the value:

DEFAULT:	The result is the kind of the default character type.
ASCII:	The result is the kind of the ASCII character type.
ISO_10646:	The result is the kind of the ISO/IEC 10646 UCS-4 character type.

SELECTED_INT_KIND(R)

Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$

Argument: R	Type: I
Result: I	Class: T

Note:

R must be scalar.

if a kind type parameter is not available then the value −1 is returned.

Example: I=SELECTED_INT_KIND(2)

SELECTED_REAL_KIND(*P,R,Radix*)

Returns a value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.

Argument: P and R	Type: I
Result: I	Class: T

Note:

1. P and R must be scalar.
2. The value −1 is returned if the precision is not available, the value −2 if the exponent range is not available, and −3 if neither is available.

Example: I=SELECTED_REAL_KIND(P,R)

SET_EXPONENT(X,I)

Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

Argument: X	Type: R
Result: As X	Class: E

I is of integer type.

Example: Exp_Part=SET_EXPONENT(X,I)

SHAPE(Source)

Returns the shape of the array argument or scalar.

Argument: Source	Type: Any
Result: I	Class: I

Note:

1. Source may be array valued or scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It must not be an assumed-size array.
2. The result is an array of rank 1 whose size is equal to the rank of Source.

Example: S=SHAPE(A(2:5,-1:1)) yields S=(4,3)

SHIFTA (I, SHIFT)

The result has the value obtained by shifting the bits of I to the right by SHIFT bits and replicating the leftmost bit of I in the left SHIFT bits.

Arguments: I	Type: I
Argument: SHIFT	Type: I (non-negative and <= BIT_SIZE(I)).
Result: Same as I	Class: E

Example: SHIFTA (IBSET (0, BIT_SIZE (0) -1), 2) is equal to SHIFTL (7, BIT_SIZE (0) - 3).

SHIFTL (I, SHIFT)
Left shift. Returns the bits of I shifted left.

Arguments: I	Type: I
Argument: SHIFT	Type: I (non-negative and <= BIT_SIZE(I)).
Result: same as I	Class: E

Example: SHIFTL (4, 1) is 8
SHIFTR (I, SHIFT)
Right shift. Returns the bits of I shifted right.

Arguments: I	Type: I
Argument: SHIFT	Type: I (non-negative and <= BIT_SIZE(I)).
Result: same as I	Class: E

Example: SHIFTR (4, 1) is 2.
SIGN(A,B)
Absolute value of A times the sign of B.

Argument: A	Type: I, R
Result: As A	Class: E

Note:

In the special case where B is zero normally the result would have the value ABS(A), but if B is one of the real kind types and the processor is able to distinguish between plus zero and minus zero then the result is ABS(A) if B is plus zero and the result is –ABS(A) if B is minus zero.
B is as A.

Example: A=SIGN(A,B)
SIN(X)
Sine.

Argument: X	Type: R, C
Result: As argument	Class: E

Note: The argument is in radians.
Example: Z=SIN(X)
SINH(X)
Hyperbolic sine.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Z=SINH(X)
SIZE(Array,*Dim*)

Returns the extent of an array along a specified dimension or the total number of elements in an array.

Argument: Array	Type: Any
Result: I	Class: I

Note:

1. Array must be an array. It must not be a pointer that is disassociated or an allocatable array that is not allocated. if Array is an assumed-size array Dim must be present with a value less than the rank of Array.
2. Dim must be scalar and in the range $1 \leq \text{Dim} \leq n$ where n is the rank of Array.
3. result is equal to the extent of dimension Dim of Array, or if Dim is absent, the total number of elements of Array.

Example: A=SIZE(Array)

SPACING(X)

Returns the absolute spacing of model numbers near the argument value. See Chap. 5 and the real numeric model.

Argument: X	Type: R
Result: As X	Class: E

Example: S=SPACING(X)

SPREAD(Source,Dim,N_Copies)

Creates an array with an additional dimension, replicating the values in the original array.

Argument: Source	Type: Any
Result: As Source	Class: T

Note:

1. Source may be array valued or scalar, with rank less than 7.
2. Dim must be scalar and in the range $1 \leq \text{Dim} \leq n+1$ where n is the rank of Source.
3. N_Copies must be scalar.
4. The result is an array of rank n+1.

Example:

if A is the array (2,3,4) then SPREAD(A,DIM=1,NCOPIES=3) then the result is the array $\begin{pmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{pmatrix}$

SQRT(X)
Square root.

Argument: X	Type: R, C
Result: As argument	Class: E

Example A=SQRT(B)
STORAGE_SIZE (A, *KIND*)
Storage size in bits. Returns the size, in bits, that would be taken in memory by an array element with the dynamic type of A.

Argument: A	Type: scalar or array of any type.
Argument: KIND (optional)	
Result: I	Class: I

Example: STORAGE_SIZE (1.0) has the same value as the named constant NUMERIC_STORAGE_SIZE in the intrinsic module ISO_FORTRAN_ENV.
SUM(Array,*Dim,Mask*)
Returns the sum of all elements of Array along the dimension Dim corresponding to the true elements of Mask.

Argument: Array	Type: N
Result: As Array	Class: T

Note:

1. Array must be an array.
2. $1 \le \text{Dim} \le n$ where n is the rank of Array.
3. Mask must be conformable with Array.
4. result is scalar if Dim is absent, or Array has rank 1, otherwise the result is an array of rank n–1.

Example:

1. SUM((/1,2,3/)) the result is 6.
2. SUM(C,Mask=C > 0.0) forms the arithmetic sum of the positive elements of C.
3. if $B = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

SUM(B,Dim=1) is (3,7,11)
SUM(B,Dim=2) is (9,12)
SYSTEM_CLOCK(*Count,Count_Rate,Count_Max*)

Returns integer data from a real time clock.

Argument: Count	Type: I
Result: N/A	Class: S

Note:

1. Count is intent(out) and is set to a processor dependent value based on the current value of the processor clock or to -HUGE(0) if there is no clock. 0 ≤ Count ≤ Count_Max.
2. Count_Rate is intent(out) and it is set to the number of processor clock counts per second, or zero if there is no clock.
3. Count_max is intent(out) and is set to the maximum value that Count can have or to zero if there is no clock.

Example: call SYSTEM_CLOCK(C,R,M)
TAN(X)
Tangent.

Argument: X	Type: R,C
Result: As argument	Class: E

Note: X must be in radians.

Example: Y=TAN(X)
TANH(X)
Hyperbolic tangent.

Argument: X	Type: R,C
Result: As argument	Class: E

Example: Y=TANH(X)
THIS_IMAGE ()
THIS_IMAGE (COARRAY,*DIM*)
Cosubscript(s) for this image.

Argument: COARRAY	Shall be a coarray of any type. If it is allocatable it shall be allocated.
Argument: DIM (optional)	Shall be a default integer scalar. Its value shall be in the range 1 <=DIM <= n, where n is the corank of COARRAY. The corresponding actual argument shall not be an optional dummy argument.
Class: T	

Results:

case (i) The result of THIS_IMAGE ( ) is a scalar with a value equal to the index of the invoking image.
case (ii) The result of THIS_IMAGE (COARRAY) is the sequence of cosubscript values for COARRAY that would specify the invoking image.
case (iii) The result of THIS_IMAGE (COARRAY, DIM) is the value of cosubscript DIM in the sequence of cosubscript values for COARRAY that would specify the invoking image.

Examples:
integer, dimension(10,20), codimension[2,0:9,0:*] :: A
integer, codimension [0:*] :: IA
for image 5:
this_image(IA) == 4 and this_image(A) = [1, 2, 0]
for image 96:
this_image(A) == [2, 7, 4]
TINY(X)
Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

Argument: X	Type: R
Result: As X	Class: I

Example: T=TINY(X)
TRAILZ (I)
Number of trailing zero bits. If all of the bits of I are zero, the result value is BIT_SIZE (I). Otherwise, the result value is the position of the rightmost 1 bit in I.

Argument: I	Type: I
Result: I	Class: E

Example: TRAILZ(4) has the value 2.
TRANSFER(Source,Mold, *Size*)
Returns a result with a physical representation identical to that of Source, but interpreted with the type and type parameters of Mold.

Argument: Source	Type: Any
Result: As Mold	Class: T

Warning: A thorough understanding of the implementation specific internal representation of the data types involved is necessary for successful use of this function. Consult the documentation that accompanies the compiler that you work with before using this function.
TRANSPOSE(Matrix)
Transposes an array of rank 2.

Argument: Matrix	Type: Any
Result: As argument	Class: T

Note: Matrix must be of rank 2. if its shape is (n,m) then the resultant matrix has shape (m,n).
Example: For $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ TRANSPOSE(A) yields $\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$

TRIM(String)
Returns the argument with trailing blanks removed.

Argument: String	Type: S
Result: As String	Class: T

Note: String must be a scalar.
Example: T_S=TRIM(S)
UBOUND(Array,*Dim*)
Returns all the upper bounds of an array or a specified upper bound.

Argument: Array	Type: Any
Result: I	Class: I

Note:
$1 \leq Dim \leq n,$ where n is the rank of Array. The result is scalar if Dim is present otherwise the result is an array of rank 1 and size n.
the result is a scalar if Dim is present otherwise is an array of rank 1, and size n.

Example: Z=UBOUND(A)
UCOBOUND (COARRAY, *DIM, KIND*)
Upper cobound(s) of a coarray.

Argument: COARRAY	Type: co
Argument: DIM (optional)	Type: I
Argument: KIND (optional)	Type: I
Result: I	Class: I

Example:
If NUM_IMAGES() == 24
INTEGER, CODIMENSION[:,:], ALLOCATABLE::A
ALLOCATE(A[1:10,*])
UCBOUND (A) is [10,3] and UCOBOUND(A,DIM=2) is 3
UNPACK(Vector,Mask,Field)
Unpacks an array of rank 1 into an array under the control of a mask.

Argument: Vector	Type: Any
Result: As Vector	Class: T

Note:
1. Vector must have rank 1. Its size must be at least t, where t is the number of true elements in Mask.
2. Mask must be array valued.
3. Field must be conformable with Mask. Result is an array with the same shape as Mask.

Example:

With *Vector* = (1,2,3) and *Mask* = $\begin{pmatrix} F & T & F \\ T & F & F \\ F & F & T \end{pmatrix}$ and *Field* = $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

The result is $\begin{pmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$

VERIFY(String,Set,*Back*)

Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

Argument: String	Type: S
Result: I	Class: E

Note:

1. The default is to scan from the left, and will only be from the right when Back is present and has the value TRUE.
2. The value of the result is zero if each character in String is in Set, or if String has zero length.
   Example: I=VERIFY(String,Set)

Appendix D

English and Latin Texts

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.

Appendix E

Coded Text Extract

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 ENBGY LU ENTTBU; OH YAN VALMN HBYOLH, HL LYANU UNMOD OG MNZY LZ YAN XOGDOFMOHN LZ TNLTUBFAJ.

Appendix F

Formal Syntax

Statement ordering

format statements may appear anywhere between the use statement and the contains statement.

The following table summarises the usage of the various statements within individual scoping units.

Kind of scoping unit	Main program	module	external sub program	module sub program	Internal sub program	interface body
use	Y	Y	Y	Y	Y	Y
format	Y	N	Y	Y	Y	N
Misc Dec[a]	Y	Y	Y	Y	Y	Y
Derived type definition	Y	Y	Y	Y	Y	Y
interface block	Y	Y	Y	Y	Y	Y
Executable statement	Y	N	Y	Y	Y	N
contains Y	Y	Y	Y	Y	N	

[a]Misc Dec (Miscellaneous declaration) are parameter statements, implicit statements, type declaration statements and specification statements

Syntax summary of some frequently used Fortran constructs

The following provides simple syntactical definitions of some of the more frequently used parts of Fortran.

Main program

```
program [ program-name ]
   [ specification-construct ] ...
   [ executable-construct ] ...
   [contains
   [ internal procedure ] ... ]
end [ program [ program-name ] ]
```

Subprogram

```
procedure heading
   [ specification-construct ] ...
   [ executable-construct ] ...
   [contains
   [ internal procedure ] ... ]
procedure ending
```

module

```
module name
   [ specification-construct ] ...
   [contains
   subprogram
   [ subprogram ] ... ]
end [ module [ module-name ]
```

Internal procedure

```
procedure heading
   [ specification construct ] ...
   [ executable construct ] ...
procedure ending
```

procedure heading

```
[ recursive ] [ type specification ] function
function-name &
   ( [ dummy argument list ] ) [ result ( result name
) ]
[ recursive ] subroutine subroutine name &
   [ ( [ dummy argument list ] ) ]
```

procedure ending

```
end [ function [ function name ] ]
end [ subroutine [ subroutine name ] ]
```

Specification construct

```
derived type definition
interface block
specification statement
```

Derived type definition

```
type [ [ , access specification ] :: ] type name
   [ private ]
   [ sequence ]
   [ type specification [[ , pointer ] :: ] component
```

```
specification list ]
   ...
   end type [ type name ]
```

interface block

```
interface [ generic specification ]
    [ procedure heading
      [ specification construct ] ...
   procedure ending ] ...
   [ module procedure module procedure name list ] ...
end interface
```

Specification statement

```
allocatable [ :: ] allocatable array list
dimension array dimension list
external external name list
format ( [ format specification list ] )
implicit implicit specification
intent ( intent specification ) :: dummy argument name
list
intrinsic intrinsic procedure name list
optional [ :: ] optional object list
parameter ( named constant definition list )
pointer [ :: ] pointer name list
public [ [ :: ] module entity name list ]
private[ [ :: ] module entity name list ]
save[ [ :: ] saved object list ]
target [ :: ] target name list
use module name [ , rename list ]
use  module name , only : [ access list ]
type specification [ [ , attribute specification ] ...
:: &
      object declaration list
```

type specification

```
integer [ ( [ KIND= ] kind parameter ) ]
real[ ( [ KIND= ] kind parameter ) ]
complex[ ( [ KIND= ] kind parameter ) ]
character[ ( [ KIND= ] kind parameter ) ]
character[ ( [ KIND= ] kind parameter ) ] &
   [ LEN= ] length parameter )
LOGICAL[ ( [ KIND= ] kind parameter ) ]
type ( type name )
```

Attribute specification

```
allocatable
dimension ( array specification )
external
intent ( intent specification )
intrinsic
optional
parameter
pointer
private
public
save
target
```

Executable construct

```
action statement
case construct
do construct
if construct
where construct
```

Action statement

```
allocate ( allocation list ) [ ,STAT= scalar integer
variable ] )
call subroutinename [ ( [ actual argument specification
list] ) ]
close ( close specification list )
cycle [ do construct name ]
deallocate( name list ) [ , STAT= scalar integer
variable ] )
endfile external file unit
exit [ do construct name ]
goto label
if ( scalar logical expression ) action statement
inquire ( inquire specification  list ) [ output item
list ]
nullify ( pointer object list )
open ( connect specification list )
print format [ , output item list ]
read (i/o control specification list ) [ input item
list ]
read format [ , output item list ]
return [ scalar integer expression ]
```

```
rewind ( position specification list )
stop [ access code ]
where ( array logical expression ) array assignment
expression
write ( i/o control specification list ) [ output item
list ]
pointer variable => target expression
variable = expression
```

Appendix G

Compiler Options

In this appendix we look at some of compiler options we have used during the development of the programs in the book.

Cray

This compiler was available on the Hector Service and several compilers are available and the default is the Portland Group compiler. To make the Cray compiler available one had to use the following commands
module swap PrgEnv-pgi PrdEnv-cray
and then
ftn -h caf compiler options source files -o executable

gfortran

gfortran

-W
-Wall
-fbounds-check
-pedantic-errors
-std=f2003
-Wunderflow -O
-fbacktrace
-ffpe-trap=zero,overflow,underflow
-fopenmp
-g

g95

g95

```
-Wall
-std=f2003
-fbounds-check
-ftrace
```

IBM

(Power 7)
xlf

```
-qlanglvl=2003pure
-qxlf2003=polymorphic
```

Intel

ifort
/check: all
Determines whether several run time conditions are checked. keyword: all, none, [no]arg_temp_created, [no]bounds, [no]format, [no]output_conversion
/coarray
/debug: full
Determines the type of debugging information generated by the compiler in the object file. keyword: minimal, partial, full, none.
/exe:% 1intel
/[no]fltconsistency
Determines whether improved floating point consistency is used.
/fpe:0
Specifies floating point exception handling at run time for the main program; n = 0, 1, or 3. 0 - floating underflow results in zero; all other floating point exceptions abort execution
/gen-interfaces
/heap-arrays

```
/inline: all
/list
```

/[no]map[:name]
Determines whether the compiler generates a link map (optionally, named name).
/O
/openmp /Qopenmp
/parallel /Qparallel

/Qcoarray
/Qfp-stack-check
/recursive
/stand: f03 /std03
/[no]traceback
Specifies whether the compiler should generate extra information in the object file that allows the display of source file traceback information at run time when a severe error occurs.
/warn: all

/tcheck
/traceback
/warn:all,nodec,interfaces

Nag

nagfor
-C=all
Compile code with all possible run time checks. all array calls do none present pointer
-C=undefined
-f2003
-info
-g
-gline
include line number information in run time error messages.
-ieee=stop
Enables all IEEE arithmetic facilities except for nonstop arithmetic. Execution is terminated on floating overflow, divide by zero or invalid operand.

sun

f95
-ansi
-w4
-xcheck=%all
-C
-ftrap=common,overflow,underflow

Index

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*,
DOI 10.1007/978-0-85729-233-9, © Springer-Verlag London Limited 2012